UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences

Architecture d'un système d'exploitation
Outils et méthodologie

L. Franck et O. Markowitch (première version en 1998)
J. Goossens, C. Macq et O. Markowitch (révision annuelle depuis 1999)
C. Macq et G. Othmezouri (révision 2002-2003)

Table des matières

Chapitre 1

Méthodologie et analyse

1.1 Introduction

Depuis quelques années, une méthodologie a été introduite au TP d'architecture d'un système d'exploitation. Il en a été décidé ainsi afin que les personnes qui suivent le TP acquièrent une démarche plus systématique lors de la conception de systèmes.

La méthodologie suit un schéma *waterfall* classique : analyse, conception implémentation et test. Les puristes pourront s'acharner à démontrer les hérésies présentes dans la démarche (comme par exemple l'absence de modèle de données conceptuel).

La bibliographie fournie en fin de syllabus reprend les ouvrages qui ont été utilisés pour élaborer ce TP ainsi que d'autres ouvrages considérés comme références pour ce TP. Ces derniers sont marqués d'une astérisque et valent la peine qu'on y jette un coup d'oeil.

1.2 L'approche méthodologique comme mode de vie (du projet)

La découpe d'un projet en phases permet, d'une part de diviser la complexité du problème traité (aspect horizontal), et d'autre part de bien mettre en évidence les relations d'ordre entre les différentes étapes de résolution (aspect vertical). L'approche utilisée est archi-classique et décompose le projet en 4 phases :

L'analyse : elle va permettre de définir le problème. On parlera d'entités de traitement, de messages et d'acteurs. Il est important de noter que tous les termes techniques sont à proscrire. On parle en terme de concepts présents dans la réalité que l'on souhaite modéliser. La phase d'analyse est généralement divisée en une phase de pré-étude et une phase d'analyse à proprement parler. Nous nous focaliserons sur l'analyse proprement dite.

La conception : elle va définir l'implantation (ici on peut parler un jargon technique) de l'analyse effectuée supra. Les choix techniques seront réalisés ici.

L'implémentation : il s'agit de la mise en musique informatique, le codage.

Les tests : ils permettent de valider le travail réalisé.

En plus de ces quatre étapes, on ajoute une phase de gestion de projet qui a lieu durant tout le projet. Elle permet de vérifier que les choses se passent bien comme prévu et selon les plans établis. Pour terminer, il y a deux phases finales sur lesquelles nous passerons car elles sortent du contexte académique : ce sont les phases de réception du produit et de maintenance. Ceci ne doit pas éclipser pour autant leur importance. Si l'étape de réception était effectuée plus strictement, bon nombre de mauvaises surprises seraient évitées et si les produits étaient facilement maintenables, leur durée de vie serait accrue, faisant chuter les coûts informatiques.

Chaque phase dans notre méthodologie (de nouveau, il s'agit d'un bien grand mot mais bon ...) va être supportée par des formalismes graphiques. Les conventions seront dûment explicitées et **il s'agira de les respecter**. En effet, à quoi bon utiliser des outils graphiques permettant de communiquer efficacement si les personnes en présence ne parlent pas le même langage.

1.3 Phase d'analyse

Elle va permettre de construire un modèle de la réalité qu'on souhaite traiter dans le projet. Modéliser la réalité implique de définir les limites de ce qu'on appelle la réalité. A titre d'exemple, dans un système de contrôle de culture, on va modéliser le suivi du taux d'humidité au moyen de capteurs. Par contre, on ne va modéliser les phénomènes de croissance des plantes. Dans cette phase d'analyse, tout terme technique (a fortiori à connotation UNIX) est déclaré tabou. Les concepts de base suivants sont définis :

Un acteur : il fournit ou reçoit des informations du système. L'acteur a un status spécial dans le sens où il est biface. Une partie de lui même interagit avec le système, cette partie apparaît dans le diagramme des entités. Une autre partie de lui même n'est pas modélisée car elle ne fait pas partie du système. Par exemple : un capteur. La partie modélisée est celle qui transmet des informations sur le degré d'humidité. La partie non modélisée est celle qui extrait ces informations de l'environnement.

Une entité de traitement : elle réalise un traitement sur de l'information. Par exemple : trier lettre.

Un message : il s'agit d'une information conceptuelle (ex : tarif d'affranchissement).

1.3.1 Le diagramme des entités

On va représenter dans le *diagramme des entités*[1] les entités de traitement au moyen d'un ovale, à l'intérieur duquel on note le nom de l'entité, ce nom contient un verbe. Un message est représenté au moyen d'une flèche (il y a donc un sens au message) entre deux entités de traitement ou entre une entité et un acteur. Sur la flèche on indique un ensemble d'informations conceptuelles séparées par des virgules. Un acteur est représenté au moyen d'un rectangle. Les acteurs sont identifiés par un nom commun. Les acteurs sont les seuls éléments du système qui peuvent uniquement recevoir ou uniquement émettre des messages. Les entités de traitement, par contre, doivent toujours avoir au moins un message entrant et un message sortant. Un acteur peut aussi être une zone de stockage d'informations.

Chaque flèche doit être perçue comme un tube destiné à transporter un et un seul type de message. Ces messages ne circulent pas en permanence, autrement dit, les "tubes" ne sont pas toujours remplis.

Pour qu'une entité de traitement puisse s'activer, il n'est pas nécessaire que tous les tubes entrants soient remplis, un seul message peut suffire à activer l'entité de traitement à laquelle il est transmit. Dans tous les cas, si toutes les entrées sont 'vides', c-à-d pas de message, l'entité ne peut être activée.

Une entité de traitement ne doit pas remplir tous les tubes qui en sort, cependant les tubes remplis le sont tous en même temps. Typiquement un centre de tri reçoit, par exemple, un message en entrée et n'active qu'un seul tube de sortie à la fois, ceci en fonction du tri.

On peut classer les entités de traitement en plusieurs catégories : entités de transformation d'information (un tube vers un tube), de multiplexage (plusieurs tubes vers un tube), de démultiplexage (un tube vers plusieurs tubes en même temps) et finalement de tri (un tube vers plusieurs tubes mutuellement exclusifs).

Il est préconisé de ne pas combiner dans une même entité plusieurs fonctionnalités distinctes (voir exception plus bas). Un conseil est de séparer les aspects communications du reste de la modélisation.

On commence l'analyse par un diagramme des entités de niveau 0. Ce diagramme sert à représenter les échanges entre le système modélisé et le monde extérieur. Le système y est donc représenté par une seule entité de traitement. Une exception à cette règle peut être envisagée s'il est important de faire apparaître une séparation entre différentes parties du système (par exemple dans le cas d'une architecture client-serveur pour laquelle le système fonctionnera

[1] Pour ceux qui connaissent les DFD, pas de comparaison hâtive.

sur deux machines différentes). Ce diagramme contient donc une ou deux (et rarement plus) entités de traitements, un ensemble d'acteurs représentants le monde extérieur interagissant avec le système, et des messages modélisant ces interactions et les échanges entre les éventuelles différentes parties du système. Pour améliorer la lisibilité de ce diagramme on peut utiliser la notion de protocole définie plus bas.

Vient ensuite le diagramme des entités de niveau 1. Ce diagramme permet de détailler de manière précise le fonctionnement interne du système. Il est construit à partir du diagramme de niveau 0. Il reprend l'ensemble des acteurs de ce niveau ainsi que l'ensemble des messages échangés avec ceux-ci. On divise toutefois les protocoles explicitement, c.-à-d. que l'on représente l'ensemble des messages constituant ces protocoles. La différence fondamentale avec le niveau 0 provient du fait que l'on y détaille tous les traitements nécessaire au bon fonctionnement du système à modéliser. On remplace donc la ou les entités de traitement, représentant le système, par un ensemble d'entités de traitement représentant des opérations simples ne nécessitant pas d'être disséquées plus en détail. On y ajoute les messages échangés entre toutes ces différentes entités leur permettant de réaliser les traitements qu'elles symbolisent. On peut également rajouter des zones de stockage si le projet nécessite de stocker ou sauver des informations (ce qui est généralement le cas). Ces zones de stockage sont représentées de la même manière que les acteurs par un rectangle.

Il est évident que le choix de la granularité, c.-à-d. de la complexité des traitements, dans le diagramme de niveau 1 dépend fortement de l'expérience des personnes réalisant l'analyse mais aussi de la complexité globale du système. Il se peut donc qu'il soit nécessaire de détailler une ou plusieurs entités de traitement du diagramme de niveau 1. Pour ce faire, on procède d'une manière similaire au passage du niveau 0 au niveau 1. On reprend donc l'ensemble des entités de traitement et des acteurs communiquant avec l'entité à détailler, et l'ensemble des messages échangés avec cette entité. On découpe ensuite l'entité de traitement en un ensemble d'entités, de messages et de zones de stockage. Ce processus peut-être appliqué récursivement sur plusieurs niveau pour des projets énormes mais nous ne serons pas confrontés à ce genre de problèmes dans le cadre du TP.

On veillera avec une attention particulière à maintenir une cohérence entre les diagrammes, ainsi un acteur ou un message, entrant ou sortant du système, ne pourra subitement apparaître ou disparaître à un niveau.

1.3.2 Notion de protocole

Notre vie de tous les jours est régie par l'utilisation de protocoles. Dans les conversations, ce sont les schémas de question/réponse, ou encore le mouvement d'une tête indiquant que l'auditeur comprend, qu'il est attentif. Lorsqu'on utilise le téléphone, on effectue une séquence déterminée d'actions : saisir le combiné, composer le numéro, attendre la sonnerie de confirmation, attendre que le correspondant décroche etc ... Ces protocoles existant dans la réalité, ils vont se trouver tout naturellement dans la modélisation effectuée au travers du diagramme des entités.

Un protocole peut donc être défini comme un ensemble de messages qui, dans le diagramme d'entités, caractérise une conversation entre deux entités de traitements. Chacun de ces protocoles sera détaillé de la manière suivante :
- Les deux entités impliquées.
- L'ensemble des messages et leur composition (en terme d'informations).
- Le séquencement de ces messages : dans quel ordre ils sont envoyés et reçus, et ce pour les deux entités impliquées.

Afin d'alléger le diagramme des entités, les messages appartenant à un même protocole entre deux entités, **et uniquement ceux-ci**, seront rassemblés sur une seule flèche bidirectionnelle.

Chapitre 2

Diagramme des processus

2.1 Méthodologie : phase de conception

La phase précédente a identifié les entités de traitement du système ainsi que les messages circulant entre ces entités. Le diagramme des entités que l'on en a déduit achève cette (brève) analyse. Il faut à présent effectuer une projection de la réalité ainsi modélisée vers un système informatique. Dans le cadre de ces travaux pratiques, l'architecture choisie est le système d'exploitation UNIX. Ce dernier permet d'effectuer des tâches de manière concurrente. Ces tâches, sont appellées processus. Nous verrons également dans un TP ultérieur que UNIX fournit des moyens permettant, pour plusieurs processus de communiquer.

Dans la phase de conception, nous allon effectuer la correspondance entre entités de traitement et processus UNIX. Les échanges entre les entités de traitement seront implantés de manière différente suivant que les entités se trouvent dans un même processus ou non. L'aspect échange d'information sera couvert dans les TPs suivants. Cette séance ci va étudier les aspects concernant la répartition des entités de traitement sur des processus UNIX.

Afin de réaliser cette répartition, nous partons du diagramme des entités. Les différentes entités vont être régroupées au sein de processus. Les critères de regroupement sont les suivants :
- Des entités devant s'exécuter en parallèle ne peuvent se trouver dans le même processus.
- Des entités se trouvant potentiellement sur des machines différentes ne peuvent se trouver dans le même processus.

On voit que malgré ces critères, il existe plusieurs répartitions possibles. La solution algorithmique de ce problème nécessite un backtracking. On se contentera ici de le faire de manière instinctive. Si nous consultons la littérature concernant les méthodologies, il existe toujours un passage où l'auteur avoue que le degré d'expérience de l'analyste est un facteur important pour telle ou telle phase. Voilà ce contrat rempli ici aussi. En effet, l'expérience aide à choisir la bonne répartition en processus.

Il est également important de noter qu'une entité peut être dupliquée dans plusieurs processus. Cela arrive lorsqu'une entité de traitement peut recevoir des messages de plusieurs entités et que ces entités sont réparties sur des processus UNIX différents.

L'outil graphique utilisé pour cette phase s'appelle le diagramme des processus. Pour le construire, nous prennons le diagramme des entités et on entoure les entités appartenant à un même processus. Par après, on dessine le diagramme des processus à proprement parlé. Les processus sont des ovales. Dans chaque ovale on note le nom du procesus (souligné) et la liste des entités qu'il regroupe. Les processus s'échangent également des messages. Ces messages sont ceux qui sont échangés entre les entités appartenant au processus et les entités appartenant aux autres processus. On constate après la découpe en processus qu'il y a deux types d'entités :

1. Les entités internes : elles ne communiquent qu'avec des entités appartenant au même processus. Leurs messages n'apparaissent pas dans le diagramme des processus.
2. Les entités externes : elles communiquent avec des entités situées dans d'autres processus (c-à-d avec d'autres entités externes). Leurs messages apparaissent dans le diagramme des processus.

Le désavantage du diagramme des processus est qu'il masque les entités internes. On peut se débarasser de cet inconvénient en utilisant un diagramme des processus à deux niveaux. Le second niveau consiste à représenter un seul processus à la fois ainsi que toutes les entités qu'il contient. Les messages de/vers l'extérieur sont représentés par des flèches avec une extrémité dans le vide. Plus simplement, ce diagramme est un diagramme des entités ou les regroupements d'entités sont séparés sur des diagrammes differents. On représente un processus par un ovale contenant des rectangles aux coins arrondis (les entités). Le nom du processus est inscrit (et souligné) en haut de l'ovale. Cette vision à deux niveaux du diagramme de procesus permet de disposer d'une vue synthétique ou d'une vue détaillée selon les besoins.

Les messages utilisent la même convention que dans le cas du diagramme des entités : des flèches avec les noms de messages. Ici aussi on préférera utiliser une aggrégation des messages sous forme de protocoles.

Le dernier type d'élément devant être incorporé dans la conception technique est l'acteur. Dépendant du problème que l'on étudie, un acteur est un processus à part entière ou est incorporé au processus comprenant l'entité avec laquelle il communique. Il existe cependant une catégorie d'acteurs méritant une attention toute particulière. Ces acteurs que l'on peut qualifier de 'réentrants', répondent à une problématique qui suit un schéma caractéristique. Un système distribué est souvent constitué d'un ensemble de sous-systèmes identiques inter-agissant ensemble. Dans le diagramme des entités, on ne représente qu'un seul de ces sous-systèmes. Les autres sont représentés par un ou des externes selon qu'il est important ou non de faire la distinction. Cette relation entre plusieurs sous-systèmes du même type doit également apparaître dans le diagramme de processus. Il faut prendre garde à bien indiquer les échanges de messages adéquats.

Une fois le diagramme des processus tracé, il faut établir les relations père-fils. Pour rappel, dans un système UNIX, chaque processus (excepté le processus init(0)) est le fils d'un autre processus. On utilisera pour cela des flèches unidirectionnnelles en pointillés qui vont du père vers le fils. Au même titre que pour les entités, on retrouve deux catégories de processus :

1. Les processus "point d'entrée". Ils sont fils d'un shell.
2. Les processus internes : ils sont crées par des processus internes ou point d'entrée.

On veillera à ce que chaque processus soit assigné à un de ces groupes.

Ces relations père/fils ne sont pas définitives. On verra que durant la réalisation du diagramme décrivant les communications entre processus il sera nécessaire de modifier ces liens de parenté. Il y aura même des cas ou certains processus devront être dupliqués. Cette cassure dans le déroulement séquentiel de la méthodologie est due au fait que le choix des moyens de communication entre processus dépend fortement des relations père/fils. Il en résulte une itération entre la phase de découpe en processus (c-à-d cette phase) et la suivante identifiant les moyens de communication entre processus. Des informations supplémentaires seront fournies au cours de cette phase.

Chapitre 3

Le diagramme des IPC

Dans le précédent diagramme, on a regroupé des entités en processus. Or, il est courant que deux entités situées dans des processus différents doivent s'échanger des informations. Le but du diagramme des IPC (Inter-Process Communication, un terme à prendre au sens large) est de définir les moyens de communications entre les différents processus. UNIX fournit un certain nombre de moyens de communication inter-processus :

- les pipes nommés et anonymes ;
- les sockets en mode datagram et stream ;
- les messages queues ;
- les signaux ;
- les fichier et mémoires partagées ;
- les sémaphores.

L'échange d'informations par fichier et mémoire partagée (shared memory) est représenté par un carré relié aux processus avec des doubles flèches ($\Rightarrow$). Une flèche vers le carré indique l'écriture vers la zone partagée, une flèche partant du carré indique une lecture. Lors d'une phase ultérieure (le diagramme d'implantation des structures de données) on spécifiera l'organisation interne du fichier ou de la zone de mémoire partagée.

Le diagramme des IPC est construit en complétant le diagramme des processus à l'aide des conventions suivantes (les processus communiquant ensemble sont reliés par des flèches) :

- Communication par pipe anonyme : un tuyau, accompagné de la lettre A.
- Communication par pipe nommé : un tuyau, accompagné de la lettre N.
- Communication par socket datagram : une flèche épaisse avec un 'D', bidirectionnelle.
- Communication par socket stream : une flèche épaisse avec un 'S', bidirectionnelle.
- Communication par message queue : une boîte contenant des messages de priorités différentes, un processus émettant un message possède une flèche dirigée vers cette boîte. Un processus lisant ces messages possède une flèche partant de cette boîte et arrivant vers lui.

- Communication par fichier : un rectangle avec un des petits côtés ouvert et le petit côté présent dupliqué. Idem pour les flèches que pour les messages queues.
- Communication par mémoire partagée : un simple rectangle. A nouveau, un processus qui lit cette mémoire possède une flèche dirigée vers lui, originaire de la boîte. Un processus écrivant dans une telle mémoire possède une flèche se dirigeant vers cette boîte.
- Envoi d'un signal : une flèche en forme d'éclair dans une boîte, comme label d'une flèche du processus envoyant le signal au processus qui le reçoit.
- Synchronisation par sémaphore : un petit drapeau dans une boîte. Les flèches sont bidirectionnelles pour les processus utilisant ce type de synchronisation.

Ces conventions permettent de représenter les différentes communications entre les processus.

La figure 3.1 synthétise ces divers moyens de communication entre processus.

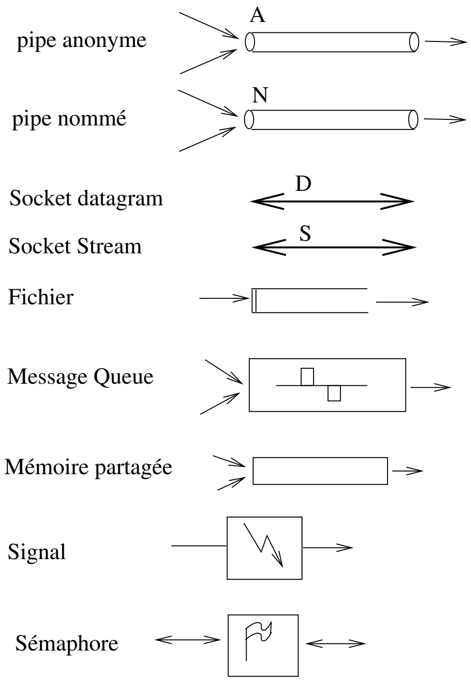


FIG. 3.1 – Conventions utilisées dans les diagrammes des IPC.

A partir du diagramme des IPC, on vérifie que :
- Pour tous les processus communicant ensemble (cf le diagramme des processus), il existe un moyen de communication défini dans le diagramme.
- Pour toutes les communications représentées dans le diagramme des IPC, les contraintes de communication (ex : relation père/fils) sont respectées.

A l'aide du diagramme des processus et du diagramme des IPC, on a terminé une des étapes importantes du design : le design architectural. On va ensuite procéder au design fonctionnel du programme : il permet d'identifier les primitives de traitement et les structures de données existantes.

Chapitre 4

Dictionnaire des traitements et des structures

Le précédent diagramme avait mené à l'énumération des différents processus du système. Chaque processus regroupe les entités identifiées lors de la première phase de l'analyse. Il faut à présent élaborer des spécifications plus précises et plus détaillées.

A partir des différents process que vous avez identifiés dans le diagramme des process, vous allez construire le dictionnaire des traitements et des structures.

Ce dictionnaire est une base de données des traitements qui seront réalisés dans chaque processus. Les structures de données seront également notées dans le dictionnnaire.

La grammaire régissant la syntaxe dictionnaire est la suivante :

dictionnaire	→ processus dictionnaire | processus
processus	→ nom-de-processus primitives structures-de-données
primitives	→ primitive primitives | primitive
structures-de-données	→ structure-de-données structures-de-données | structure-de-données
primitive	→ nom-de-primitive description-rôle-primitive

structure-de-données → nom-de-structure
type-de-structure
description-rôle-structure

nom-de-processus → *chaîne de caractères*

nom-de-primitive → *chaîne de caractères*

description-rôle-primitive → *chaîne de caractères*

nom-de-structure → *chaîne de caractères*

description-rôle-structure → *chaîne de caractères*

type-de-structure → *"table (triée)"* |
"liste (triée)" |
"arbre ((trié)" |
"delta-liste" |
"hash-table" |
"graphe" |
"record"

Une primitive est une fonction qui rend des services. Ces services sont soit internes (ils sont uniquement utilisés par le processus) ou externes (ils sont utilisés indirectement par un autre processus).

Exemple : dans un processus serveur de pages HTML (un serveur WWW), il y a les primitives *recevoir requête du réseau*, *envoyer réponse sur réseau*, *lire page HTML sur le disque*. Il y a aussi des primitives du type *ouvrir accès réseau*, *fermer accès réseau*, etc

Dans cet exemple, *lire page HTML sur le disque* est un service interne tandis que *recevoir requête du réseau* est externe car il est 'en quelque sorte' invoqué par le browser émettant la requête.

Il est important de choisir une bonne granularité de découpe. Cela revient à choisir des primitives ni trop simples, ni trop complexes. Une primitive de taille adéquate est une primitive dont ont perçoit précisement le fonctionnement interne sans que ce dernier ne soit trival.

L'identification des structures de données découle naturellement de l'identification des primitives. On fournira une description du rôle de la structure ainsi que son type (arbre, arbre trié, ...).

Une fois le dictionnaire completé, une phase de vérification est nécessaire. Elle permet de s'assurer que toutes les primitives nécessaires sont présentes et que toutes les primitives identifiées sont utiles.

La première vérification se réalise en effectuant des traces dans le diagramme des entités. Une trace est un exemple d'éxécution du système. Elle commence à partir d'un acteur et plus particulièrement d'un acteur émettant des messages vers le système. La réaction du système à ce message va être décrite en énumérant les différentes primitives invoquées lors du traitement de la requête. Lorsque le système revient dans un état 'inactif', c'est-à-dire lorsque la requête de l'utilisateur a été satisfaite, la trace est terminée. Pour de gros systèmes, faire la trace de toutes les requêtes peut s'avérer très lourd car le nombre de traces possibles est très grand. Ceci est dû non seulement au grand nombre de requêtes mais également aux multiples cas possibles pour une requête. Dans de tels cas, on s'efforce de dégager un sous-ensemble de traces telle que la couverture du système soit maximale. Une méthode parmi d'autres consiste à éliminer des traces qui sont incluses dans d'autres. Les cas limites doivent bénéficier d'une attention toute particulière. Finalement, il ne faut pas tomber dans le piège qui consiste à définir des primitives trop complexes (trop grandes) afin de simplifier la réalisation des traces.

La seconde vérification consiste à s'assurer que chaque primitive est présente dans au moins une trace.

Chapitre 5

Diagramme d'implantation des traitements

5.1 Introduction

Au cours de la précédente étape de la méthodologie (le dictionnaire des traitements et des structures), on a identifié une liste de primitives et de structures de données. L'étape actuelle (le diagramme d'implantation des traitements) va préciser et organiser le contenu de cette liste. Les structures de données seront, elles, abordées lors de l'étape suivante.

Préciser chaque primitive revient à fournir un nom d'implantation, détailler les paramètres d'entrée, l'information fournie en retour et les pré/post conditions. Organiser le dictionnaire consiste à regrouper en modules les primitives qui, dans un même processus, agissent sur des concepts identiques.

5.2 Spécification des primitives

Pour chaque primitive identifiée dans le dictionnaire, on va ajouter les informations suivantes :

- La liste des paramètres fournis à la primitive (paramètres entrants).
- La liste des paramètres que la primitive renvoit (paramètres sortants).
- Les conditions dans lesquelles la primitive peut être appelée (précondition).
- Les conditions que la primitive garanti après son appel (postcondition).
- La liste des codes de retours fournis par la primitive.

Un paramètre est formé d'un nom et d'un type. Le type est soit un type simple (entier, flottant, booléen, chaîne de caractère) soit un type composé. Les types composés *doivent* faire référence à une structure de données répertoriées dans le dictionnaire. Remarquez que le mode de passage (par valeur ou par référence) n'est pas abordé ici. Cet aspect est dépendant du langage de programmation. Par contre, à partir de cette étape, il faut prendre en compte les deux aspects

suivants :
- Tous les noms doivent prendre une forme "programmable" (en fonction du langage utilisé). Il faut pour cela adopter des conventions de nommage (exemple : " LireSocket" ou "afficher_texte").
- Les noms 'globaux' tels que les noms de structures de données doivent être figés et utilisés de manière homogène dans tous les documents.

Les conditions dans lesquelles une primitive peut être appelée se nomment les *préconditions*. Elles définissent les limites du contrat, comment la primitive doit se comporter selon ses spécifications fonctionnelles (c-à-d qu'elle doit réaliser sa tâche). Quelques exemples :
- Pour une primitive `addToTable(Element elm)` une précondition est que la table soit correctement initialisée.
- Pour une primitive `dispatch(Element elm)` une condition est que le champs discriminateur de `elm` prenne uniquement des valeurs permettant la discrimination.

De la même manière, les conditions garanties par la primitive après son appel sont appelées *postconditions*. A titre d'exemple, si on considère une primitive `translate(Element elm)`, la postcondition exprime les transformations appliquées à `elm`. Les postcondition ne sont, souvent, qu'une retranscription du rôle de la primitive.

Finalement, les codes de retour fournissent une liste d'identificateurs (également en forme "programmable") indiquant les codes utilisés pour exprimer le status de l'exécution de la primitive. Le nommage de ces codes doit suivre une convention claire et doit être homogène. Par exemple, toutes les primitives ajoutant un élément à une structure de données sont susceptibles de retourner le même code d'erreur indiquant qu'il n'y a plus de place.

Toutes ces informations vont être rajoutées au dictionnaire des traitements et des structures. Les ajouts à apporter à la définition de la grammaire sont les suivants :

primitive	→	nom-de-primitive description-rôle-primitive nom-d-implantation paramètres-entrants paramètres-sortants préconditions postconditions codes-d-erreur
paramètres-entrants	→	paramètre-entrant paramètres-entrants | paramètre-entrant
paramètre-entrant	→	nom-paramètre type-paramètre

paramètres-sortants	$\rightarrow$ paramètre-sortant paramètres-sortants | paramètre-sortant
paramètre-sortant	$\rightarrow$ nom-paramètre type-paramètre
préconditions	$\rightarrow$ précondition préconditions | précondition
postconditions	$\rightarrow$ postcondition postconditions | postcondition
codes-d-erreur	$\rightarrow$ code-d-erreur codes-d-erreur | code-d-erreur
code-d-erreur	$\rightarrow$ identificateur-d-erreur cause-d-erreur
nom-d-implantation	$\rightarrow$ *chaîne de caractères*
nom-paramètre	$\rightarrow$ *chaîne de caractères*
type-paramètre	$\rightarrow$ *chaîne de caractères*
précondition	$\rightarrow$ *chaîne de caractères*
postcondition	$\rightarrow$ *chaîne de caractères*
identificateur-d-erreur	$\rightarrow$ *chaîne de caractères*
cause-d-erreur	$\rightarrow$ *chaîne de caractères*

5.3 Organisation en modules

Une fois les primtives correctement identifiées, il faut à présent les rassembler en groupes logiques. Un tel groupe est appelé *module* (ou librairie, package, boîte à outils, ...). Un module contient des primitives provenant d'un même processus. Ces primitives agissent toutes, soit sur la même structure de données, soit elles sont toutes des sous-tâches d'une réalisation plus globale. A titre d'exemple, toutes les primitives permettant d'accéder à une table de hashing doivent être placées dans un même module. Toutes les primitives commandant un robot pour le montage d'une roue sur une voiture (ex : VisserBoulonHautGauche, VisserBoulonBasDroit, ...) doivent être dans le même module. Chaque module doit porter un nom (ex : TableHashing, MontageRoue).

Il n'est pas impossible qu'un même module soit utilisé dans plusieurs processus. De tels cas doivent être correctement identifiés. Il est hors de question de créer deux modules différents dont les primitives sont identiques.

Une fois les différents modules identifiés, il faut dessiner le diagramme de modules. Ce dernier permet d'établir les relations existant entre les modules (représentés sous la forme de rectangles, le nom du module est inscrit à l'intérieur). Ces relations sont la cause directe du concept de trace. Pour rappel, une trace est un enchaînement de primitives. Chaque primitive appartenant dorénavant à un module, on peut également voir ces enchaînements au niveau des modules. Lorsque une primitive d'un module fait appel à une primitive d'un autre module, on relie les deux modules à l'aide d'une flèche. La relation établie est appelée une relation *utilise* (comme un appel de fonctions, un "call") et la flèche est placée près du module utilisé. Une fois le diagramme en modules tracé, il faut vérifier la présence de cycle. S'il existe un cycle, la découpe en module (la répartition des primitives dans les modules) est incorrecte. L'idée est de présenter le diagramme en modules sous forme de couches. La couche du bas représente les fonctionnalités de plus bas niveau, utilisant directement l'OS, et chaque couche supérieure successive, représente un nouveau degré d'abstraction par rapport à l'OS (un module d'une couche intermédiaire peut parler de communication sans plus préciser la nature de son implémentation). Le nombre de couches dépend de la complexité du système.

Le dictionnaire est à nouveau modifier pour y décrire les modules.

dictionnaire	→ entrée-dico dictionnaire | entrée-dico
entrée-dico	→ processus | module
module	→ primitives-module
primitives-module	→ nom-d-implantation primitives-module | nom-d-implantation

Chapitre 6

Diagramme d'implantation des structures de données

6.1 Introduction

Lors du précédent TP, le dictionnaire des traitements a été étendu afin d'inclure des informations détaillées à propos des primitives formant les processus. Ensuite, ces primitives ont été groupées en modules. Un module contient des primitives qui, soit travaillent sur les mêmes concepts de données, soit participent à la même tâche globale.

6.2 Le diagramme des structures de données

La partie traitement étant à présent complètement spécifiée, il faut spécifier également la partie données. Cette étape est divisée en deux tâches qui ne sont pas à effectuer séquentiellement :

- Identifier les primitives relatives à la création, l'utilisation et la destruction des structures de données. Ces primitives devront être détaillées de la même manière que les primitives de nature plus "fonctionnelle". Il est important de bien faire la distinction entre primitives de base d'accès à une structure et les primitives plus abstraites. Si on prend l'exemple d'un agenda, identifier la primitive `AjouterRendezVous` ne suffit pas. Il faut également définir la primitive `AjouterElmDansListe` si l'agenda est implanté sous forme d'une liste.
- Spécifier l'implantation de la structure de données. Il s'agit de préciser si la structure sera partagée ou non , la manière dont le partage sera effectué et finalement, quelle est l'organisation interne de la structure en elle même.

6.2.1 Structure partagée ou non

Il est souvent nécessaire que plusieurs processus disposent des mêmes informations. Dans de tels cas, plusieurs solutions sont possibles pour réaliser ce partage : soit on duplique l'information, soit on fournit un accès partagé à l'information. La duplication de l'information pose des problèmes de synchronisation qui sont souvent fort coûteux. D'un autre côté, la charge (en terme de traitement) introduite par la consultation de l'information peut être distribuée sur plusieurs processus et/ou machines. Cependant, à moins que la solution distribuée ne soit vraiment indispensable, on lui préfère une solution où les données ne sont pas dupliquées.

Dans ce cas, deux schémas sont possibles :
- Les procédures d'accès sont centralisées, ce qui mène à un schéma de type client/serveur. La liste suivante reprend des architectures de type client/serveur de complexité croissante.
  - Une table protégée par des primitives d'accès. Le client et le serveur sont situés dans des processus différents sur la même machine. La communication s'effectue au moyen des IPC et pipes vus précédemment.
  - Une table protégée par des primitives d'accès. Le client et le serveur sont sur des machines différentes. La communication s'effectue au travers d'un réseau. Ceci constitue le cas le plus fréquent lorsque l'on pense architecture client/serveur.
- L'autre schéma, lorsque les données ne sont pas dupliquées, consiste à utiliser une mémoire accessible à partir de plusieurs processus. Ce schéma peut être implanté grâce aux shared memory. Ce type d'architecture permet de paralléliser au maximum les accès, cependant beaucoup de précautions doivent être prises pour gérer adéquatement les accès concurrents de manière à conserver l'intégrité de la structure de données. Dans ce but, on utilisera souvent des sémaphores.

6.2.2 Accès à une structure partagée

Suivant que l'on se trouve dans un schéma client/serveur ou non, différents aspects doivent être pris en compte concernant l'accès aux données.

Dans un cas client/serveur, l'absence d'accès concurrents rend le schéma d'accès très simple. Par ailleurs, si les requêtes peuvent être bufferisées au niveau du serveur, une politique cohérente de service doit être mise sur pied pour éviter les cas de famine. Il faudra également spécifier clairement quelle est la part du travail réalisée par le client et quelle est la part du travail du serveur. Déplacer de l'intelligence vers le client permet d'alléger la tâche du serveur mais exige plus de prérequis au niveau du client. Il s'agit avant tout de décisions de design.

Dans des schémas non client/serveur, les accès concurrents modifiant la structure de données doivent être séquentialisés. Les accès en lecture peuvent être gardés concurrents. Ces derniers ne peuvent de toute façon pas être effectués lorsque la structure est en cours de modification.

6.2.3 Organisation interne d'une structure de données

Cette question est particulièrement intéressante lorsqu'il faut implanter des structures non triviales telles que des tables de hashing ou des graphes. La difficulté consiste à marier les contraintes d'espace et de temps d'accès qui sont généralement antagonistes.

6.3 Extension du dictionnaire

On ajoutera les informations suivantes au dictionnaire :
- Un nom d'implantation pour chaque structure.
- Les primitives de gestion des structures de données.
- Les modules regroupant ces primitives (le diagramme en modules doit être modifié en conséquence).
- Pour chaque structure, une explication détaillée de l'organisation interne de la structure et des méthodes d'accès.

En résumé :

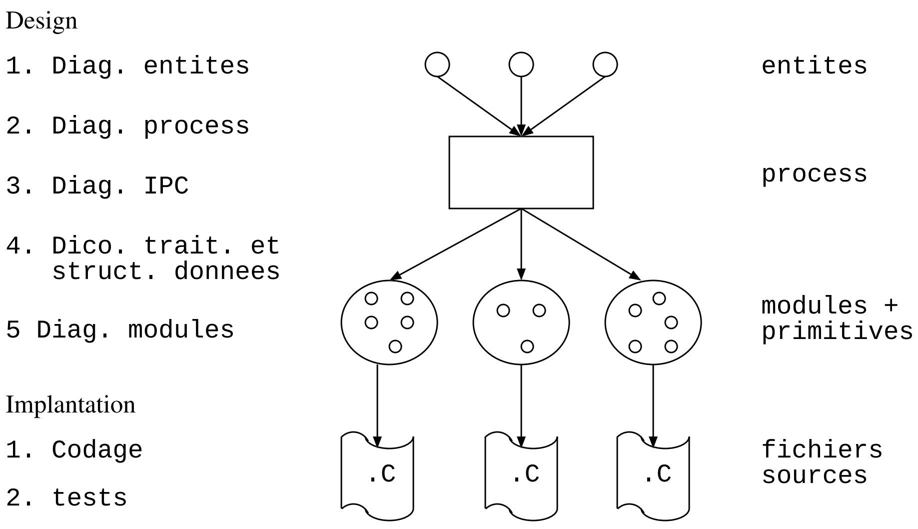
structure-de-données	→	nom-de-structure nom-d-implantation type-de-structure description-rôle-structure primitives-structure description-organisation-structure description-implantation-structure
primitives-structure	→	nom-primitive primitives-structure | nom-primitive
description-organisation-structure	→	*chaîne de caractères*
description-implantation-structure	→	*chaîne de caractères*

Chapitre 7

Planning et suivi de projet

Ce chapitre traite de l'organisation du travail lors de la réalisation d'un projet.

Un projet se divise en une phase de design et une phase d'implantation :



Il faut essayer d'évaluer le temps de réalisation du projet et organiser cette réalisation.

Dés le début du projet, on évalue pour chaque entité (du diagramme des entités) le temps de design (étapes 1 à 5). Ce temps est évalué en un certain nombre de demi-jours (une demi-journée = 4h/homme). On dit alors que la *granularité* est la demi-journée. Si des entités semblent pouvoir être « designées » en moins d'une demi-journée, alors on les regroupe toutes et on évalue le temps de design de la totalité en demi-journée(s).

Finalement, la somme de toutes ces durées donne la durée totale de la phase de design. Cette durée totale est non compressible (on n'y travaille pas en parallèle dans le cadre de notre projet).

Il faut alors tenter d'évaluer le temps nécessaire à la phase d'implantation. Le problème est que l'implantation peut se faire avec des tâches réalisées en parallèle (quand plusieurs personnes travaillent ensemble).

Le temps de développement peut être évalué ainsi :

$$T_{development} = \frac{T_{design} \cdot K}{Nombre\ de\ personnes} + S$$

Où :

K = le facteur d'homogénéité qui dépend du projet (s'il est peu ou beaucoup parallélisable). Ce facteur est usuellement compris entre 1 et 2.

S = le temps de synchronisation entre les différentes personnes. Ce temps dépend du projet et du nombre de personnes. Ce temps s'exprime est un certain nombre de demi-jours.

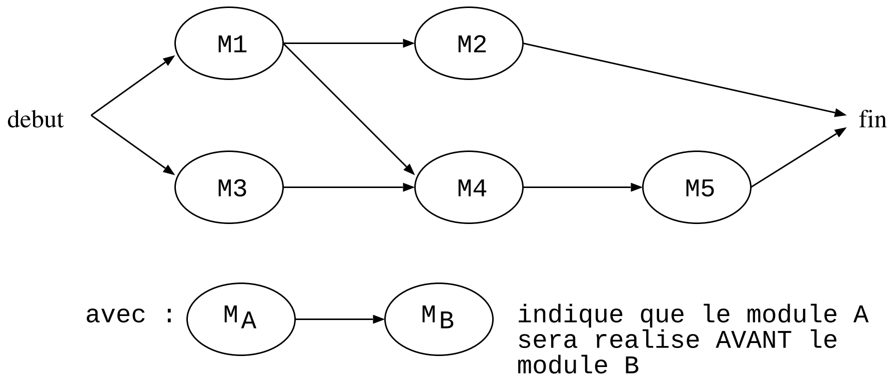
Il n'y a malheureusement pas de méthode déterministe pour connaître les valeurs de K et S, seule l'expérience permet d'affiner l'évaluation de ces valeurs.

Sur base du temps de design et d'implantation, il est possible de prévoir la date de fin du projet et de prévoir le coût de celui-ci.

Dés que l'evaluation du temps est effectuée, on continue le design (process, IPC, traitements, structure de données, modules et primitives). A ce niveau (avant l'implantation) il faut faire un planning pour l'implantation. Ce planning servira à gérer le travail des différentes personnes et permettra le suivi du projet. Pour ce faire nous allons partir des primitives récemment identifiées et raisonner comme pour les entités. On part d'un module et on évalue la complexité de chaque primitive du module. Cette complexité est aussi exprimée en un certain nombre de demi-journée(s). On regroupe les primitives ayant un complexité inférieure à une demi-journée, en évitant d'en rassembler plus de 3 (ou 4). Pour chaque paquet de ces primitives (qui ont chacune une complexité inférieure à une demi-journée) on exprime la complexité en demi- journée(s). Si une primitive a une complexité plus grande qu'une semaine, il faut alors l'éclater en plusieurs primitives (cela signifie que le découpage n'a pas été assez fin).

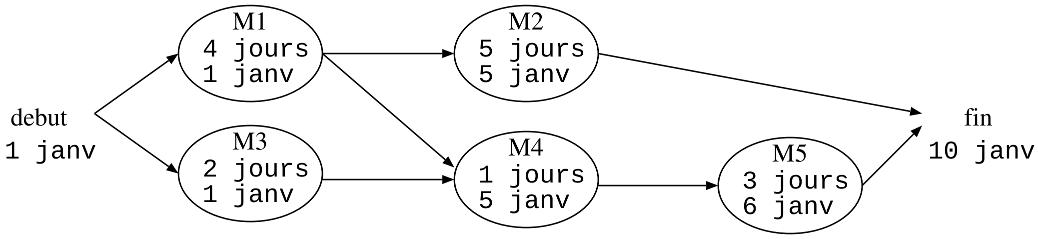
Dés que la complexité de toutes les primitives a été évaluée, on peut déterminer le temps d'implantation de chaque module. Mais il faut que dans ce temps de développement soit repris le temps nécessaire aux différents tests (tests unitaires, d'intégration, fonctionnel, reviewing). Le temps nécessaire au développement d'un module ainsi qu'à ses tests est égal au temps de développement du module préalablement établi + (le temps de développement du module préalablement établi divisé par 2).

Sur base des complexités des modules nous allons réaliser un planning. Le premier outil est le *graphe de précédence* (Pert) dans lequel tous les modules apparaissent et où nous mettons en évidence que tel module doit être réalisé avant tel autre module :

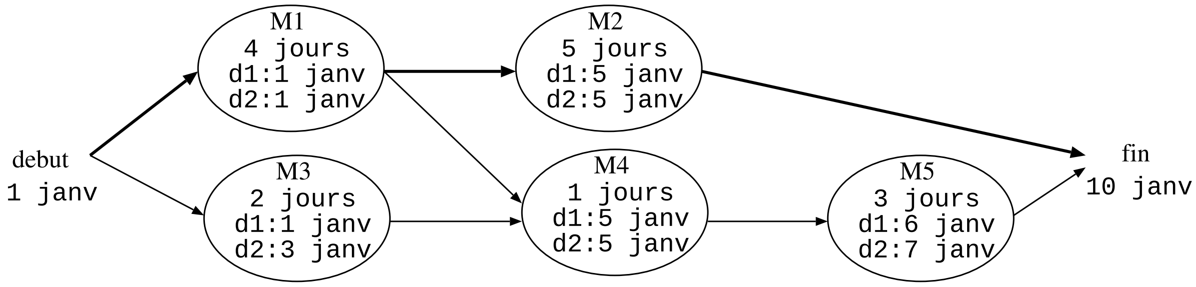


Nous voyons que certains modules peuvent être réalisés simultanément.

Au moyen de ce graphe de précédence et en utilisant les complexités calculées pour chaque module, nous pouvons fixer la date de fin du projet (en parcourant le graphe de gauche à droite) :



Puis en parcourant le graphe de droite à gauche nous pouvons déterminer la date à laquelle un module doit être commencé au plus tard :

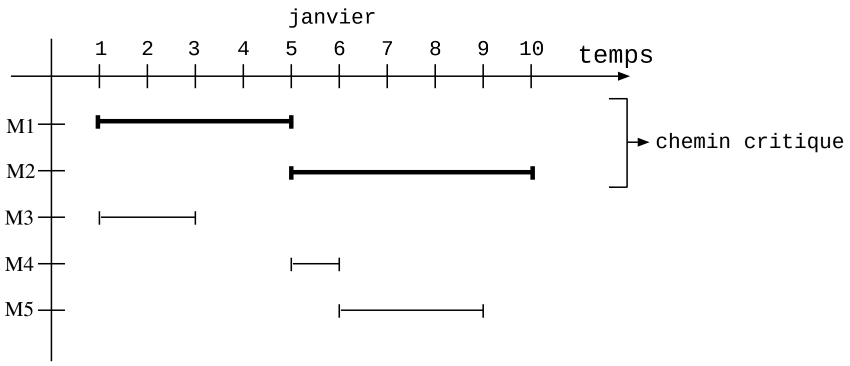


Avec $d1$: date au plus tôt ; et $d2$: date au plus tard

Nous voyons que « début - M1 - M2 - fin » est un *chemin critique* car tous les modules de ce chemin ont la date de début égale à la date de début au plus tard. Un chemin critique est tel que si du retard est pris sur un de ses modules alors c'est tout le projet qui prend du retard.

A ce moment, il est intéressant (et nécessaire) de comparer la complexité totale de développement évaluée à cette étape (en connaissant la date de fin du projet) avec celle évaluée au début du design. Il est aussi significatif de comparer le temps prévu au début du design pour réaliser le design avec le temps réellement utilisé.

A partir des infos du graphe de précédence, on peut tracer le graphe Gant qui est plus lisible :



On utilise ici les dates « au plus tôt ».

Avec ce graphe nous voyons exactement les dates de début et de fin. Le suivi du projet peut se faire aisément (identification des retards). Nous voyons facilement ce qui peut être réalisé en parallèle.

Enfin après le développement, il est à nouveau intéressant de comparer le temps réellement utilisé pour réaliser le développement avec les différents temps évalués en cours de travail (avant le design et avant l'implantation).

Chapitre 8

Processus, fichiers, appels systèmes

8.1 Généralités

Il existe deux courants : le system V de AT&T et BSD de Berkeley.
Chez AT&T on trouve le début de Unix. Puis on note une prise de vigueur dans les unifs. Il exite aussi une volonté d'uniformisation : POSIX.

Durant le TP nous préciserons, lorsque nous parlerons de system-call, si nous somme en Syst V ou en BSD.

8.2 Les processus

Pour introduire le fonctionnement de Unix, nous allons dans un premier temps nous intéresser au noyau : le kernel. Le kernel est la partie qui se trouve entre les programmes user et le hardware. Le kernel est le centre nerveux d'un OS (Unix décentralise beaucoup de commandes). Sur une machine tournant Unix, nous voyons trois couches : user programs / kernel / hardware. Le kernel gère entre-autre les processus.

Un processus (process) est une instance d'un programme qui s'exécute. On parle d'instance car on peut avoir plusieurs instances d'un même programme (ce qui donnera plusieurs process).

Il faut noter que le kernel n'est pas un ensemble particulier de process qui tournent en parallèle avec les process user. Le kernel est une partie de chaque process. Plus précisément on peut dire que les processus se partagent le kernel.

Un process est composé de plusieurs parties (segments). On trouve un segment de code ("text"), un segment de données, un stack, ainsi qu'un stack système et un segment de données système. Ces deux derniers se trouvent en dehors de l'espace d'adressage du process. On ne peut donc y apporter de modifications (ni même les lire).

Voyons maintenant quels sont les services réalisés par le kernel :
- Il contrôle l'exécution des process en permettant leur création, leur fin, leur suspension et leur communication.
- Il élit de manière équitable les process pour avoir accès au processeur (par ex. time sharing).
- Il attribue de la mémoire principale aux processus s'exécutant et il gère cette mémoire.
- Il gère la mémoire secondaire (système de fichiers hiérarchique).
- Il gère les I/O.

Voyons comment sont gérés les process.

Chaque process a une entrée qui lui correspond dans la process table (située dans le kernel). On trouve comme infos dans cette process table : l'état du process, l'id du propriétaire des process, ...

Pour chaque process, on alloue une U-Area qui contient d'autres infos comme les pointeurs vers les file descriptors des fichier ouverts par ce process, la home et la current directory, ... On alloue séparément une U-Area car ainsi on ne réserve de la mémoire qu'à la création d'un process, contrairement à la process table qui a une taille fixe.

Chaque process a un contexte : l'état du process, la valeur de ses variables globales, la structure de ses données, l'état des registres de la machine, des infos de la process table et la U-Area, le contenu de son stack et le contenu du stack système qui le concerne. Toutes ces infos sont nécessaires pour réactiver un process qui aurait été endormi ou à qui on aurait retiré le processeur.

8.3 Les fichiers

Les informations concernant un fichier se trouvent dans son inode, lequel se trouve sur disque. "Inode" vient de Index Node.

Tout fichier a un et un seul inode. Un fichier peut avoir plusieurs noms, mais tous ces noms réfèrenceront le même inode. Chaque nom sera un "link".

Les inodes sont sous forme statique sur disque et quand on ouvre un fichier le kernel les lit et les recopie dans l'inode mémoire pour les gérer. Les modifications apportées au fichier seront reportées (si nécessaire) dans l'inode mémoire. L'inode disque ne sera mis à jour que plus tard (pour des raisons de performance).

Sur disque, l'inode décrit les attributs du fichier et la position de ses données sur le disque. On trouve donc dans l'inode disque : l'id du propriétaire du fichier, le type de fichier (normal, directory, pipe, ...), les permissions d'accès, la date de dernière modification, la date de dernier accès, le nombre de lien au fichier, la table des adresses sur disque des données du fichier et la taille du fichier.

L'inode mémoire est une copie de l'inode disque avec des infos complémentaires comme par exemple la position de l'inode disque sur le disque.

Dans les inodes, il y a donc la table des adresses sur disque des données du fichier. La structure de cette table est :

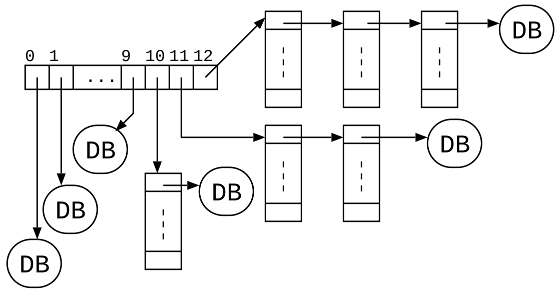


Fig. 8.1 – Table des inodes

Pour exemple, si un bloc fait 1 Kbytes et si une adresse est codée sur 4 bytes, on peut stocker 256 adresses dans un bloc d'indirection et on est à même d'adresser au total 16 Gbytes.

Pour gérer les fichiers, le kernel utilise différentes structures de données : la file table et la UFDT (user file descriptor table). On trouve dans le système une seule file table, mais chaque process a sa UFDT.

Voici la structure de gestion de fichiers ouverts. Cette structure permet de partager l'accès à des fichiers.

8.4 Les system call

Les system call permettent aux process de faire des opérations qui leur sont autrement interdites. Quand un process réalise un system call, le kernel prend alors en charge l'exécution du system call et renvoit en cas de problème un code d'erreur.

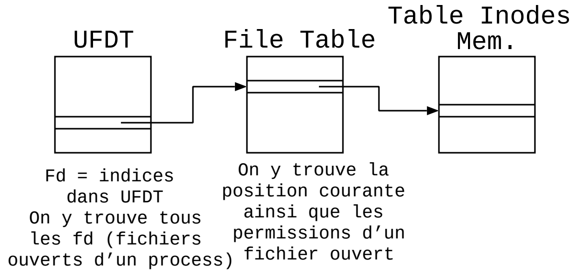


Fig. 8.2 – UFDT, FT et Inodes

La création d'un nouveau process passe aussi par une requête au kernel via un system call. Seul un process actif peut donc créer un nouveau process. Le nouveau process hérite de la plupart des attributs du process créateur (appelé le process père).

Comment cela se réalise t'il ? On crée un nouveau process à partir d'un process père lorsque ce dernier réalise le system call "fork". On trouve donc dans le code du process père la commande :

$pid = fork();$

Tous les process sont créés de cette façon sauf un. Le process 0. c'est le premier process tournant sur la machine. Ce process 0 est construit par le kernel au "boot" du système. Et à partir de ce process 0 vont etre créé via "fork" les process nécessaires au fonctionnement du système.

Voyons ce qu'il se passe lors de la réalisation d'un fork :
- Le kernel alloue un emplacement dans la process table pour le nouveau process
- On associe pour ce nouveau process (fils) un pid
- On réalise une copie du contexte du process père dans le contexte du process fils
- Le kernel incrémente les compteurs d'accès des fichiers (dans les inodes mémoires concernés) associés au process, ...
- On retourne le pid du fils au père et le 0 au fils

Remaque importante : lors du recopiage du contexte, la UFDT du père est donc recopiée dans le contexte du fils.

8.5 Autres system call

Exit : un process se termine en appelant (explicitement ou implicitement) exit. "$exit(int)$" : int est retourné au process parent.

Wait : un process peut synchroniser son exécution avec la fin d'un de ses process fils en utilisant le system call wait. "$pid = wait(adr\_etat)$" avec pid qui est le pid du fils zombie et adr_etat est l'adresse de l'espace user qui contiendra le code d'état de l'exit du fils.

Un programme exécutable est chargé en mémoire par le kernel via le system call exec. Le programme appelé va recouvrir l'espace mémoire du process appelant. Il y a plusieurs type d'exec (execl, execv, execle, execve) en fonction du type de paramètres. Le plus couramment, on utilise $execl(path + filename, filename, arg, ..., 0)$ ou les arg sont des $char*$.

Il existe aussi : creat, open, close, read, write, lseek, fnctl (exemple : $cmd = F\_setfl$ et $arg = O\_NDELAY$ ou O_APPEND, ...), ...

Dup : redirection des I/O. "$int\ dup(fd)$" : recopie le pointeur vers la file table associé à "fd" dans la première entrée libre de la UFDT.

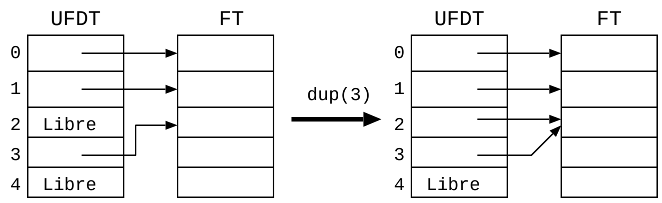


FIG. 8.3 – dup(2)

Ainsi si on fait "$close(0)$" puis "$dup(fd)$", on est sûr que le pointeur en "fd" sera recopié en stdin, on redirige donc ainsi le stdin vers un fichier ouvert connu.

Chapitre 9

Les pipes

9.1 Les pipes anonymes

Les pipes anonymes sont le premier moyen de communication inter-process que l'on va voir. Ils fournissent un canal FIFO unidirectionnel entre deux processus. Ces deux processus doivent avoir un lien de parenté (ce lien est souvent un lien direct père-fils). On appelle ces pipes "anonyme" car aucun nom de fichier ne leur est attaché. La création d'un pipe anonyme se fait en manipulant la UFDT. On utilise le system call `pipe(2)`[1]. Le paramètre transmis est un pointeur vers un tableau de deux entiers. `pipe(2)` crée le pipe et lui associe deux entrées dans la UFDT (les FD sont placés dans le tableau fourni en paramètre). Le premier FD permet la lecture dans le pipe, le second permet l'écriture. L'interface d'accès à un pipe anonyme est alors exactement la même que celle d'accès à un fichier. Certaines opérations (`lseek(2)`) sont impossibles étant donné l'aspect FIFO du pipe.

La lecture sur un pipe est décomposée en deux cas selon qu'il y ait des données à lire ou non. Lorsqu'il n'y a pas de données à lire, le system call `read(2)` se comporte de deux manières différentes pour peu que le pipe soit en mode bloquant ou non. S'il est en mode bloquant, le `read(2)` est mis en sleep jusqu'à ce qu'il y ait suffisamment de données pour satisfaire la lecture. S'il est en mode non bloquant, le system call retourne immédiatement un code d'erreur. Le mode du pipe (bloquant ou non) est spécifié par le system call `fcntl(2)`, par défaut il est ouvert en mode bloquant.

L'écriture dans un pipe qui n'a pas pas de FD ouvert en lecture génère un signal (voir plus loin) `SIGPIPE`.

Les pipes anonymes ont une limite en taille. Cette limite est spécifiée par la

[1] Le chiffre après le nom du system call renvoie aux man pages. Tous les détails sont fournis dans ces dernières.

constante `PIPE_BUF`. Sur Linux, `PIPE_BUF` vaut 4,096 bytes[2]. En dessous de cette limite, les écritures sont garanties atomiques. Cette particularité est importante si plusieurs processus écrivent simultanément sur le pipe : les messages ne seront pas inter-lacés. Dans le cas d'une écriture d'un nombre de bytes supérieur à `PIPE_BUF` ou simplement d'un nombre de bytes 'explosant' la capacité du pipe, l'intégrité des messages peut être violée. Une certaine synchronisation est donc nécessaire entre les lectures et les écritures afin d'éviter les ennuis.

9.1.1 Utilisation des pipes anonymes

Pour que deux process communiquent via un pipe anonyme, ils doivent posséder tout les deux les FD de lecture et d'écriture. Ceci ne peut se réaliser que de la manière suivante :

1. Créer les FD à l'aide de `pipe(2)` dans un des process.
2. Forker le second process (qui garde la copie de la UFDT).
3. Communiquer les numéros des FD du pipe au second process.

Les deux premières étapes ne peuvent être inversées : les FD doivent être créés avant la duplication de la UFDT. La dernière étape se réalise souvent en donnant en ligne de commande les numéros des deux FDs (lors du `exec(2)`). On peut trouver contraignant la nécessité de créer le pipe avant le fork, on peut également trouver contraignant la relation obligatoire de parenté entre les deux processus. Les pipes nommés suppriment ces contraintes.

Les pipes anonymes ne fournissant qu'un canal unidirectionnel, un canal bidirectionnel est obtenu par la création de deux pipes (un pour chaque sens de la communication). Dans une telle configuration, un process ferme le FD de lecture du pipe sur lequel il écrit. Il ferme aussi le FD d'écriture du pipe sur lequel il lit.

9.2 Les pipes nommés

Les pipes nommés sont des pipes anonymes auxquels on attribue un nom. Dès lors, les contraintes de parenté n'ont plus de raison. Avant de pouvoir être utilisé, le pipe nommé doit être créé : on utilise pour cela le system call `mknod(2)`

Ensuite le pipe nommé s'ouvre comme un fichier régulier et les system calls habituels (`open(2)`, `read(2)`, `write(2)`, `close(2)` et `unlink(2)`) peuvent être utilisés (à l'exception de `lseek(2)`...).

Etant donné que, contrairement aux pipes anonymes, un pipe nommé peut s'ouvrir en read-only ou en write-only, l'existence d'un processus lecteur n'implique pas l'existence d'un processus écrivain (et vice et versa). Les system calls d'ouverture, de lecture et d'écriture ont des comportements différents.

[2](`/usr/include/linux/limits.h`)

- L'ouverture d'un pipe en écriture qui n'a pas été ouvert en lecture par un autre processus peut entraîner une mise en sleep du processus écrivain jusqu'à ce qu'un processus lecteur ouvre le pipe (cela dépend du mode : bloquant ou non).
- L'ouverture d'un pipe en lecture qui n'a pas été ouvert en écriture par un autre processus peut entraîner une mise en sleep du processus lecteur.

La lecture et l'écriture se comportent de la même manière que dans le cas des pipes anonymes
- Lecture et écriture existent en mode bloquant et non bloquant.
- L'écriture dans un pipe qui n'a pas de lecteur génère le signal `SIGPIPE` pour le processus écrivain (ce cas est identique aux pipes anonymes).
- Lorsque le dernier processus lecteur meurt, tous les processus écrivains restant reçoivent un signal `SIGPIPE`.

Les pipes nommés sont sujet aux mêmes contraintes que les pipes anonymes. Pour terminer, la littérature anglophone appelle les pipes anonymes *pipe* et les pipes nommés *FIFO*.

9.3 Introduction aux IPCs

Les IPC (InterProcess Communication) sont des services qui ont été introduits dans la release 4 de System V. On distingue trois sortes d'IPC :
- Les message queues : des files de messages accessibles en FIFO ou aléatoirement.
- Les sémaphores : des drapeaux avec compteur dont la manipulation est atomique.
- Les mémoires partagées : des zones de mémoire partagées entre plusieurs process.

Lorsqu'une message queue, un sémaphore ou une mémoire partagée est créée par un process, la mort de ce process n'implique pas l'effacement de l'IPC → attention de ne pas encombrer le système avec des IPC obsolètes. Les commandes (shell) `ipcs` et `ipcrm` permettent d'une part d'examiner le statut des IPC dans le système et d'autre part d'effacer une IPC existante.

Chapitre 10

Les message queues

La message queue est une file de messages accessibles par plusieurs process. Les messages sont composés d'une partie data et d'un type (un entier). Le type du message (d'usage libre) permet d'accéder à la queue autrement qu'en FIFO.

La message queue est créée à l'aide du system call `msgget(2)`. Les message queues sont identifiées par une clé. La connaissance de la clé est nécessaire pour tous les process voulant accéder à la message queue. Des permissions semblables à celles des fichiers sont possibles. On utilise dans ce cas l'UID et le GID du processus.

Le system call `msgsnd(2)` permet d'envoyer des messages. Pour rappel le message est décomposé en une partie type (un entier) et une partie data proprement dite. La réception d'un message se fait grâce au system call `msgrcv(2)` (la lecture est destructive). Il faut spécifier la message queue que l'on désire accéder et le type de message que l'on veut lire. Différentes manières d'exprimer le type permettent un accès séquentiel ou aléatoire. Les man pages et la littérature fournissent plus de détails à ce propos. Tout comme les pipes, la lecture (et l'écriture) peuvent se faire en mode bloquant ou non. Le system call (`msgctl(2)`) permet de gérer la message queue (basculer le mode bloquant/non bloquant, effacer la message queue etc...).

Il existe quelques contraintes sur les message queues :

Taille maximale d'un message : 2048 bytes.

Taille maximale d'une queue : 4096 bytes.

Nombre maximum de queues dans le système : 50.

Chapitre 11

Les signaux

Les signaux fournissent un moyen asynchrone d'avertir un process qu'un événement s'est produit. Il existe un trentaine de signaux dont la sémantique est fixée (ex : `SIGPIPE` pour la mort du dernier process lecteur d'un pipe). Les signaux `SIGUSR1` et `SIGUSR2` sont réservés pour l'utilisateur.

La mise en oeuvre des signaux est basée sur deux system calls. Le system call `signal(2)` permet d'associer une fonction de traitement (*handler*) à un signal donné. La fonction de traitement sera automatiquement invoquée lors de la réception du signal. Elle s'exécute dans le contexte du processus recevant le signal. Cette fonction peut être la fonction par défaut (utile pour désinstaller un handler précédemment installé). Dans ce cas, un comportement standard est prévu par le kernel. Ce comportement varie selon les signaux. Une autre possibilité consiste à demander que le signal soit ignoré. Le second system call est `kill(2)` qui envoie un signal à un process (identifié par son pid).

Quelques remarques doivent être faites concernant les signaux :
- Certains signaux ne peuvent être ignorés (comme `SIGKILL` causant la fin du process).
- Seul le super-user peut envoyer des signaux à tous les process. Un utilisateur ´normal´ ne peut envoyer des signaux qu'aux process dont il est le créateur.
- Lors d'un system call `exec`, la configuration des signaux est mise à sa valeur par défaut.
- Lors du traitement d'un signal, le comportement à avoir en cas de signal est positionné au traitement par défaut → la première chose à faire dans un handler est de réactiver le traitement que l'on veut pour le signal (attention aux conditions de course !).
- L'activation d'un signal pouvant avoir lieu à n'importe quel moment, un system call peut être interrompu. Certains system calls (ex : les lectures en mode bloquant) vont retourner une valeur de -1, il faudra les relancer.
- Dans la routine de traitement d'un signal, il est possible de faire appel à des system calls → attention ! Car tous les system calls ne sont pas ré-entrants.

L'envoi de signaux est également disponible à partir du shell via la commande `kill(1)`.

D'un point de vue 'interne' les signaux sont gérés dans le kernel. Ainsi dans Linux, à chaque process est associé un vecteur de bits (un bit par signal), lorsqu'un signal est reçu le bit est basculé à 1. On a également une table donnant pour chaque signal l'action à entreprendre. Finalement, on a un masque permettant de bloquer des signaux temporairement (i.e., l'action associée n'est pas réalisée MAIS le signal n'est pas ignoré, il est placé en attente). Ces structures sont vérifiées à chaque fois que le process retourne d'un system call ou est reschedulé.

L'utilisation des signaux peut entraîner quelques problèmes majeurs (ex : dans le cas de conditions de course au niveau du handler). Il est donc chaudement conseillé de manipuler les signaux avec beaucoup de prudence et de faire attention aux aspects suivants :
- Le traitement d'un signal peut-il être interrompu par un autre signal ou pire par le même signal ?
- Utilise-t-on des system calls dans le traitement du signal ?
- Un system call bloquant pourrait-il être interrompu par un signal ?

Le problème des signaux est que l'utilisateur n'a pas un contrôle ferme sur se qui se passe. A titre d'exemple, il est possible que le processus qui installe un handler pour un signal donné reçoive ce signal avant d'avoir pu effectuer l'installation du handler. Il est très difficile de se prémunir contre ce genre de failles. L'autre problème vient du fait que la gestion d'un signal n'est pas atomique : d'autres signaux peuvent survenir à n'importe quel moment. Les signaux ne sont donc à utiliser que dans les circonstances où on maîtrise complètement l'environnement.

Chapitre 12

La mémoire partagée

La mémoire partagée fait partie de la famille des IPC. Elle permet d'allouer un bloc contigu en mémoire et de l'inclure dans l'espace d'adressage de plusieurs processus. Les primitives `shmget(2)`, `shmat(2)`, `shmdt(2)` et `shmctl(2)` permettent la création, l'attachement, le détachement et le contrôle des blocs de mémoire partagée. Le bloc mémoire est alors accédé comme si il s'agissait d'une zone mémoire allouée avec `malloc(3)`. Il est important de garder à l'esprit que le bloc est attaché à une adresse différente selon le processus. Si une structure de données est stockée dans le bloc mémoire, il faut donc exprimer toutes les adresses par rapport au début du bloc et non pas de manière absolue.

Chapitre 13

Les sémaphores

Les sémaphores sont des compteurs que l'on peut incrémenter (opération V) et décrémenter (opération P) de manière atomique. Les sémaphores 'traditionnels' contiennent des valeurs binaires (0 ou 1). Les sémaphores dits généraux sont n-aires, mais toujours dans le domaine strictement positif. Une particularité intéressante des sémaphores réside dans la possibilité d'avoir une décrémentation bloquante si le compteur vaut 0 avant le début de l'opération. Le processus décrémentant (opération P) est mis en sleep jusqu'à ce que un autre processus incrémente le compteur du nombre d'unités suffisant pour autoriser l'opération P mise en sleep précédemment.

Les sémaphores sont souvent utilisés dans les routines d'allocation de ressources présentes en nombre limité dans le système. On utilise un sémaphore général dont le compteur est initialisé au nombre de ressources initiallement présentes dans le système. Un processus tentant de réserver plus de ressources que ce qu'il n'y a de disponible sera mis en sleep jusqu'à ce que d'autres processus libèrent les ressources nécessaires.

Une autre utilisation commune des sémaphores est la protection de section critique. Une section critique est un ensemble d'instructions qui ne peut être exécuté simultanément par plusieurs processus (ex : la modification de pointeurs dans une liste).

La valeur du sémaphore est initialisée avec le nombre maximum d'exécution simultanées possibles de la section critique (généralement 1). Le protocole d'entrée de la section consiste à faire un P, le protocole de sortie oblige de faire un V. De cette manière, si un processus exécute la section critique, les autres processus devront attendre (ils seront mis en sleep) que ce processus ait fini avant d'exécuter à leur tour, un par un, la section critique.

Cet exemple permet de bien comprendre l'utilité de l'aspect atomique des opérations P et V. Typiquement une opération P nécessite de tester la valeur du sémaphore et ensuite de le décrémenter si c'est possible. Si ces deux opérations ne sont pas réalisées de manière atomique, il existe des cas de P réalisés en concurrence qui résulteraient en un non respect du comportement du sémaphore.

Les system calls permettant de manipuler les sémaphores sont :

`semget(2)`, `semctl(2)` et `semop(2)`.

Il faut remarquer que l'implantation System V des sémaphores fournit des sémaphores généraux et qu'il est possible à l'aide d'un system call de modifier plusieurs sémaphores (on manipule en fait des vecteurs de sémaphores).

Chapitre 14

Les sockets

Les sockets sont une solution originellement BSD au problème consistant à concevoir des programmes pouvant utiliser les réseaux sans être attachés à un seul type de réseau.

Les sockets sont donc une interface entre les processus désirant communiquer et les protocoles existant sur les réseaux.

Les processus communiquant via les sockets utilisent généralement le modèle client-serveur. Le processus serveur étant à l'écoute d'un socket.

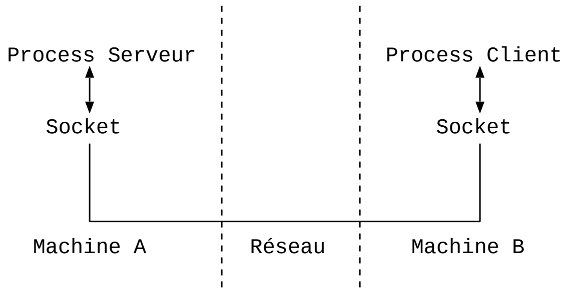


FIG. 14.1 – Socket entre deux processus.

On peut voir un réseau comme un graphe. Pour communiquer il existe deux types de communication.

Soit on établit une connexion (un chemin dans le graphe) alloué une fois pour toute entre les machines A et B, c'est ce qu'on appelle un circuit virtuel. Avec ce type de connexion, mis à part les données, circulent aussi des informations de contrôle qui permettent de s'assurer que tout ce qui a été envoyé est bien reçu.

L'autre type de communication, appelé datagramme, ne réserve pas un chemin. Nous avons donc une communication sans connexion (connection less). Ainsi chaque paquet de donnée prend un chemin qui peut être différent de celui emprunté par d'autres paquets. Il n'y a pas d'information de contrôle, il est donc possible de perdre des paquets. De plus comme le chemin suivi par les paquets peut être différent, il n'y a pas d'ordre d'arrivée des paquets qui est assuré.

Pour utiliser les sockets, il y a une série ordonnée de system calls à utiliser.

- `socket(2)` : établit un des bouts d'un lien de communication. On y spécifie le domaine de communication (sur une machine : AF_UNIX, sur Internet AF_INET, ...) On indique le type de communication (circuit virtuel sûr et coûteux : SOCK_STREAM, ou datagramme : SOCK_DGRAM).
- `bind(2)` : associe un nom (dans le cas de AF_UNIX) ou une adresse (dans le cas de AF_INET) au descripteur de socket retourné par le system call `socket`. Ce sera ce nom ou cette adresse qui sera connu par les processus client désirant communiquer avec le processus serveur ayant réalisé le system call `bind`. Dans l'adresse (AF_INET) se trouve aussi le numéro de port auquel les processus clients doivent se connecter.
- `listen(2)` : spécifie des informations sur la file d'attente (ce trouvant du côté serveur) qui stocke les requêtes de connexion venant des processus clients. On spécifie, entre-autre, le nombre maximum de requêtes en suspens.
- `accept(2)` : reçoit des requêtes en vue d'une connexion à un processus client. `accept` réagit lorsque le client fait un `connect`, sinon il s'endort. Ce system call retourne comme paramètre l'adresse du client qui se connecte (le même type d'adresse que ce qui est utilisé avec `bind`), et retourne aussi un nouveau socket descriptor qui sera celui avec lequel le serveur pourra communiquer avec le client. Tout ce que le client écrira dans son socket sera reçu via ce nouveau socket descriptor. L'idée est de faire en sorte que le serveur puisse continuer à attendre de nouvelles requêtes via son socket descriptor précédent et continuer à communiquer avec le process client via un canal séparé.
- `connect(2)` : réalise un connexion à un socket descriptor existant (de l'autre côté de la ligne de communication). On spécifie l'adresse du socket cible (adresse = nom si AF_UNIX ou = adresse IP + port si AF_INET). Les deux sockets doivent bien-entendu utiliser les mêmes domaines et les mêmes protocoles de communication.
- Une fois la communication établie les processus peuvent communiquer en utilisant les system calls `read(2)` et `write(2)` (en spécifiant les socket descriptors adéquats) ou `send(2)` et `recv(2)`.
- pour fermer une connexion de type socket, on utilise le system call `shutdown(2)` en indiquant le sens de la communication qui doit être coupé. Il faut noter que les socket descriptors ne sont pas touchés par ce system call.
- `close(2)` : libère un socket descriptor.
- `getsockname(2)` : accède au nom d'un socket lié par un `bind`.

En mode "datagramme", on communique de la manière suivante (car pas de connexion) :

- `sendto(2)` et `recvfrom(2)` : en indiquant chaque fois l'adresse du correspon-

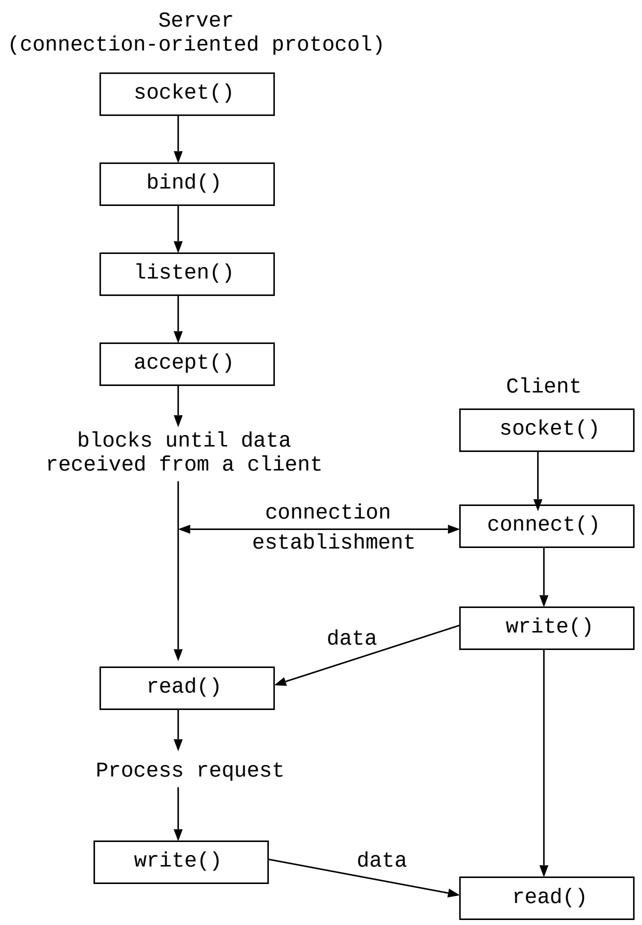


FIG. 14.2 – Communication en mode stream.

dant.

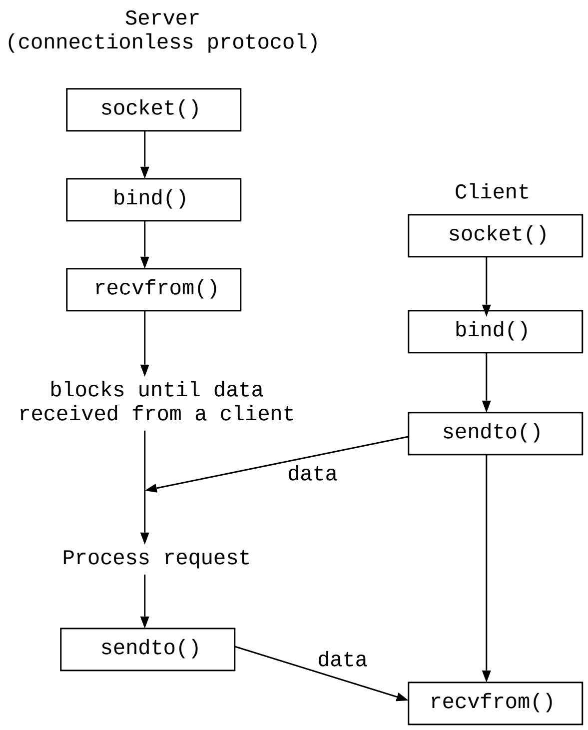


FIG. 14.3 – Communication en mode datagramme.

Remarques :
- les librairies à utiliser sont (à checker dans les man pages car dépend du système) :
  - #include<sys/types.h>
  - #include<sys/socket.h>
- Il est possible au moyen du system call `fcntl(2)` de rendre les system calls non bloquant.
- Faire attention lorsqu'on transmet des entiers (par exemple) via des sockets. En effet il faut veiller à ce que les deux machines sur lesquelles se trouvent les processus en communication, représentent bien les entiers de la même façon (position du bit de poids fort à gauche ou à droite, ...).
- Faire attention au "port" utilisé en AF_INET. Les ports de 0 à 1023 sont réservés pour le système.
- Certain signaux (SIGPIPE) sont déclenchables lors d'utilisation de sockets (voir man pages).

Chapitre 15

Gestion de la mémoire

15.1 Les types de mémoire, types d'accès

Registres < cache < RAM < mémoire secondaire
On peut accéder à la mémoire de manière séquentielle ou directe.

15.2 Introduction

Quand nous demandons de la mémoire à l'OS, nous ignorons comment ce dernier gère cette mémoire (comment il gère le heap), elle nous apparaît sous la forme d'un bloc contigu de mémoire, comme un espace de travail, comme un segment. Et ce segment que nous nous sommes alloués, c'est à nous de le gérer. En effet l'application qui va utiliser les fonctionnalités que nous réalisons pourra effectuer :

- une consultation d'une information dans cet espace mémoire,
- une modification d'une information,
- une suppression d'une information,
- un ajout d'une nouvelle information.

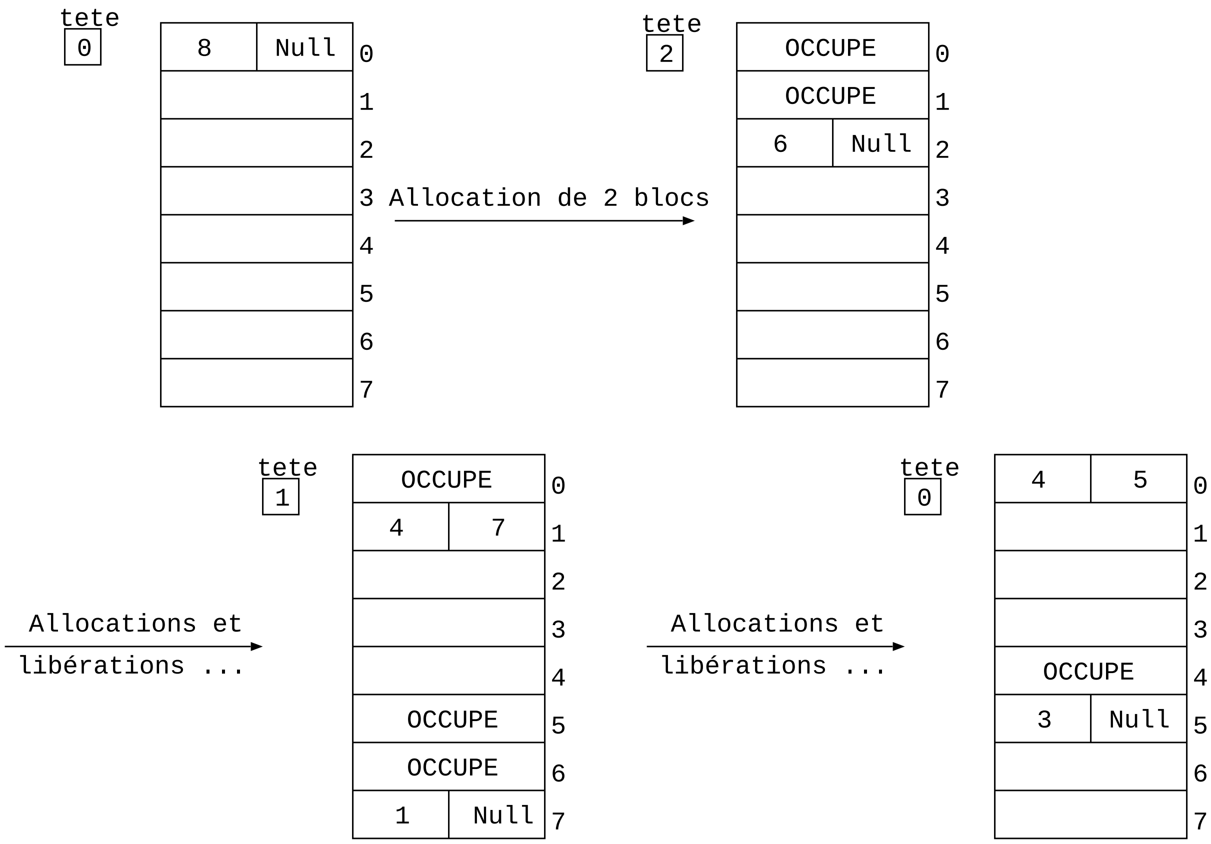
Il nous faut donc une technique de récupération des emplacements qui ne sont pas ou plus utilisés de façon à pouvoir les réutiliser.
Mettons nous dans la situation (la plus probable) où notre programme propose à une autre application deux procédures :

- allouer_emplacement(adr&, taille),
- liberer_emplacement(adr, taille).

Il existe, entre autres, deux techniques : Bitmap (tableau des occupations) et Free-list (liste d'emplacements vides).

15.3 Technique avec free-list

Une free-list est une liste qui lie les emplacements de mémoire qui sont libres (qui n'ont jamais été alloués ou qui ont été libérés). Une free list est représentée ci-dessous :



Lorque l'on gère une free-list, il faut décider l'ordre suivant lequel les blocs libres seront triés :

– on peut lier les blocs par ordre d'arrivée,
– on peut lier les blocs par ordre d'adresse,
– on peut lier les blocs par la taille des trous.

Le choix du tri a de l'importance car par soucis d'efficacité, chaque fois qu'un emplacement sera libéré, il faudra éventuellement le fusionner avec les emplacements libres qui lui sont contigus.

– Avec le tri par ordre d'arrivée, il faut parcourir toute la free-list pour détecter si une fusion est possible avec le nouveau bloc libre.
– Avec le tri par adresse, il suffit de parcourir la liste jusqu'à l'adresse du bloc qui précède le bloc que l'on vient de libérer.
– Avec le tri par taille des bloc, il faut tout comme le tri par ordre d'arrivée, parcourir la free-list.

Lorqu'une application voudra s'allouer un bloc de mémoire, il faudra chercher dans la free-list un bloc de mémoire libre de taille suffisante.

S'il n'y a pas de bloc libre de taille suffisante, on peut adopter plusieurs politiques :

- On peut mettre l'application demandeuse en attente.
- On peut forcer un tassement.
- On peut bouger de place les données de blocs non-libre (ce n'est pas toujours possible et c'est plutôt déconseillé).
- Si on a le contrôle sur les structures de données du programme utilisateur, on peut se rendre compte qu'une zone de mémoire est inaccessible et que l'on peut donc la récupérer (garbage collection).

A présent, nous rencontrons le problème de la stratégie suivant laquelle nous allons choisir un bloc libre lors d'une demande d'allocation. Il faut bien comprendre que le comportement (en terme d'efficacité) d'une stratégie dépend de la taille moyenne, maximum et minimum des blocs en présence, ainsi que de la durée durant laquelle les blocs sont conservés. C'est pourquoi une stratégie peut se révéler bonne pour un certain profil de demande et mauvaise pour un autre.
Comme stratégie d'allocation il existe entre-autres :

- first fit : on cherche un bloc libre de taille plus grande ou égale à la taille demandée,
- best fit : on cherche le plus petit bloc libre tel que la taille est plus grande ou égale à la taille demandée,
- worst fit : on prend systématiquement le bloc libre le plus grand (l'idée étant que si l'on prend le plus gros bloc pour en extraire un petit, il restera encore un bloc de taille utile. L'expérience montre que, dans de nombreux cas, cette stratégie fonctionne aussi bien que le best fit).

Pour les stratégies first et best fit, on peut, soit allouer tout le bloc, ce qui provoque de la fragmentation interne. Soit on alloue juste ce qu'il faut et on remet dans la liste ce qui reste, ceci provoque de la fragmentation externe. On peut aussi mélanger les deux en refusant de remettre des blocs plus petits qu'une certaine taille limite dans la free-list.

Avec une stratégie first fit, le choix est rapide mais produit beaucoup de fragmentation externe. Les stratégies de best et worst fit produisent moins de fragementation externe mais nécessitent soit une recherche dans toute la liste, soit de maintenir la liste triée (par taille).

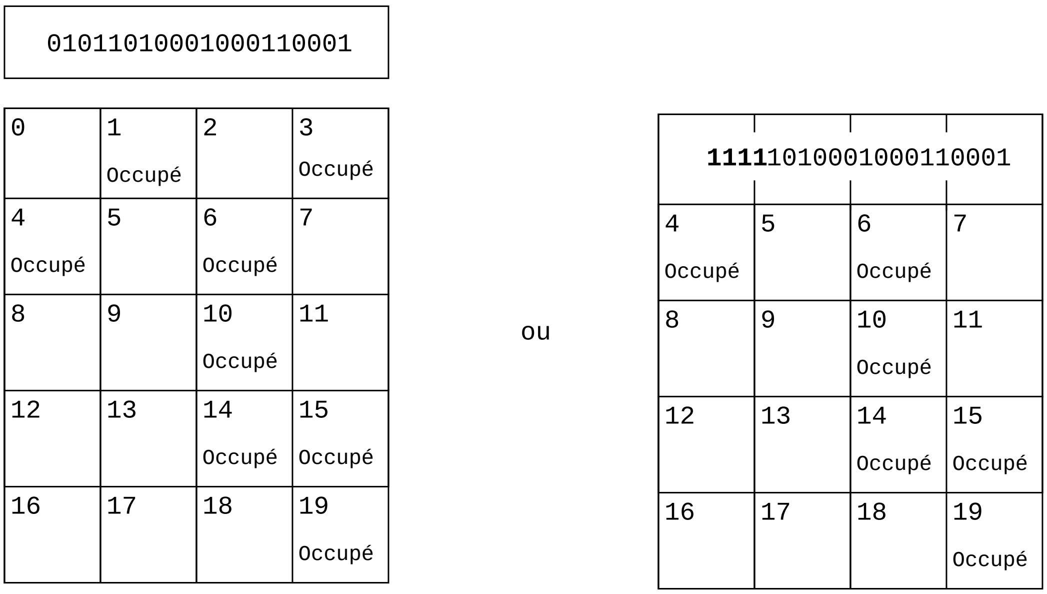
Il existe d'autres stratégies, entre autres le buddy system (ou subdivision binaire).

15.4 Technique avec bitmap

Ici aussi nous allons manipuler des blocs au sein de segments. Une bitmap est un tableau de bits. A chaque bit de la bitmap correspond un bloc dans la mémoire. Le principe de la bitmap est que chaque bit de celle-ci indique si le bloc lui correspondant est libre ou occupé. Un bit à 1 indique que le bloc est occupé, un bit à 0 indique que le bloc est libre.

Il faut dans le cas d'une gestion avec bitmap que les blocs soient de taille fixe. Sinon comme dans la bitmap nous n'avons pas d'indication quant à la taille des blocs, il nous est impossible de pouvoir allouer une certaine quantité de mémoire.

Il y a plusieurs possibilités pour implanter les bitmaps. Soit on l'implante dans une zone extérieure au segment concerné, et dans ce cas le premier bit de la bitmap donne accès au premier bloc du segment et ainsi de suite. Soit on implante la bitmap dans le segment en question ... mais dans tous les cas dans un endroit fixe et bien précis du segment. Dans ce cas, il y a encore deux possibilités : soit le premier bit de la bitmap correspond au premier bloc qui suit la bitmap (si cette dernière est en tête du segment). Soit le premier bit de la bitmap correspond au premier bloc du segment et dans ce cas, si la bitmap est en tête du segment, ce premier bloc fait partie de la bitmap et est donc occupé (par la bitmap). Dans cette situation tous les bits de la bitmap correspondant aux blocs qui contiennent la bitmap doivent être à 1 (=occupé).



Avec la bitmap les problèmes de fusion des bloc contigus n'existent plus. Ici la fusion est immédiate.

Il faut bien choisir la taille des blocs, car il faut se rendre compte que plus petits sont les blocs, plus grande est la bitmap. Exemple : si un bloc fait 4 bytes, on a 1 bit (de la bitmap) pour 4 bytes (32 bits), cela fait 1 bit sur 33 utilisé par la bitmap (1/33 de la mémoire et ce, quelle que soit la taille de la mémoire). Si un bloc = 1 byte, la bitmap occupe 1/9 de la mémoire. Il faut donc trouver un compromis entre la réduction de la taille de la bitmap et le risque de fragmentation si les blocs sont trop grands.

Un défaut de la bitmap est que lorsqu'on veut allouer k blocs de mémoire, il faut trouver k bits consécutifs à 0, c'est une opération lente et non triviale.

L'allocation des blocs se réalise avec les mêmes stratégies que celles utilisées avec la free-list à savoir first fit, best fit, worst fit, ...

En pratique, on utilise parfois une bitmap pour gérer la mémoire secondaire (exemple : la fat sur les PC tournant Dos-Win 3.x/95).

Il peut arriver que lorsqu'on agrandit un segment, la taille de la bitmap ne soit plus suffisante. Il y a alors plus de blocs que de bits dans la bitmap. Dans ce cas on utilise un partie à la fin de la bitmap pour indiquer (un pointeur vers) une suite à la bitmap.

Chapitre 16

Développement en C

16.1 Introduction

Le langage C est un des langages haut niveau offrant le plus de libertés au programmeur. Cette liberté a un prix qui se chiffre en nombre de bugs qu'il est possible de générer (à son corps défendant ...). C'est pourquoi il est indispensable d'adopter une certaine discipline de développement. Cette discipline est orientée selon deux axes : l'utilisation à bon escient des possibilités du langage et l'adoption de certains réflexes de type "préventif".

16.2 Les fichiers .h et .c

Les fichiers .h sont appelés *include files* et les fichiers .c *source files*. On place dans les fichiers .h :
- Les prototypes de fonctions.
- Les structures globales (qui doivent être visibles de l'extérieur du module).
- Les variables globales.
- Les constantes globales.

On place dans les fichiers .c :
- Les fonctions (développées).
- Les structures locales au module (c-à-d qui ne doivent pas être visibles de l'extérieur).
- Les variables locales au module.
- Les constantes locales.

A titre d'exemple, supposons que l'on implante un module permettant de manipuler des stacks d'entiers.

Le fichier .h se présente comme suit :

```
/* stacks.h */

#define STACK_SIZE 30
```

```
typedef struct Stack
{
        int stack[STACK_SIZE];
        int top;
} Stack;

void initStack(Stack *a_stack);
void pushOnStack(Stack *a_stack, int an_int);
int popFromStack(Stack *a_stack);


/**** Autre fichier ****/


/* stacks.c */

#include "stacks.h"

void initStack(Stack *a_stack)
{
  .
   .
    .
}

void pushOnStack ....
   .
    .
     .
```

Lorsqu'un module désire utiliser un stack d'entiers, il commence par inclure le fichier .h `include "stacks.h"`. Ensuite, il s'alloue un stack (statiquement : `Stack my_stack` ou dynamiquement `my_stack=(Stack *)malloc(sizeof(Stack))`. Le module peut alors manipuler le stack au travers des primitives prévues à cet effet.

La définition de la structure est placée dans le fichier .h car, comme on le constate, l'utilisateur du module Stacks doit connaître le type de cette structure pour pouvoir compiler son propre module.

Si on change légèrement cet exemple on spécifiant que le module Stacks fournit un seul stack qui est utilisé par tout le monde, la répartition .h .c devient la suivante :

```
/* stack.h */
```

```
void initStack();
void pushOnStack(int an_int);
int popFromStack();


/**** Autre fichier ****/


/* stack.c */

#include "stack.h"

#define STACK_SIZE 30

typedef struct Stack
{
        int stack[STACK_SIZE];
        int top;
} Stack;

Stack the_stack;

void initStack()
{
  .
   .
    .
}

void pushOnStack ....
   .
    .
     .
```

On constate que :

1. Les fonctions n'ont plus de paramètre `a_stack`,
2. Dès lors la définition de la structure n'est plus nécessaire dans le fichier .h,
3. Dès lors la définition de la constante n'est plus nécessaire dans le fichier .h.
4. Une variable globale est déclarée dans le fichier .c, elle constitue le paramètre implicite de toutes les fonctions manipulant le stack.

En résumé, la règle de répartition .c .h est la suivante :
- On place dans le fichier .h tout ce qui est nécessaire aux utilisateurs du module pour compiler leurs propres modules.

- On cache dans le fichier .c tout ce qui ne peut être visible de l'extérieur.
- Cette découpe doit être motivée par le fait que moins un utilisateur connaît les détails d'implantation, plus on évite les complications.

16.3 Protection contre l'inclusion multiple des .h

Supposons que, dans le fichier `source.c` on trouve inclus deux fichiers .h : `stacks.h` et `brol.h`. Par ailleurs, `brol.h` procède également à l'inclusion de `stacks.h` (car, par exemple, il est également utilisateur d'un stack). Lors de la compilation de `source.c` une erreur de compilation sera levée due à la définition multiple des primitives de `stacks.h`. Il faut donc protéger un fichier .h contre l'inclusion multiple.

Ceci est réalisé en définissant lors de la première inclusion un symbole (selon une certaine convention) qui est unique dans le scope du fichier include. Lors de chaque inclusion on testera l'existence de ce symbole. Si le symbole existe déjà, on est dans le cas d'une tentative d'inclusion multiple. Les directives de compilation suivantes permettent d'atteindre ce but. L'exemple est donné pour `stacks.h`.

```
stacks.h

#ifndef __STACKS_H
#define __STACKS_H

Corps du fichiers stacks.h comme
vu auparavant.


#endif
```

Le lecteur est invité à vérifier le comportement des primitives lors d'une tentative d'inclusion multiple.

16.4 La compilation séparée

L'utilisation de la découpe en fichiers .c .h est motivée par deux raisons. D'une part la découpe en modules est respectée. D'autre part cela permet la compilation séparée. Cette dernière consiste à produire des fichiers objets (.o) individuellement pour chaque module. Ensuite tous les fichiers objets sont linkés afin de produire l'exécutable. Ce type de compilation est particulièrement avantageux pour les gros projets : la modification d'une des sources n'entraîne pas la nécessité de recompiler tout le code.

La commande (en gcc) permettant de produire un fichier objet à partir d'un fichier .c est `gcc -c source.c`. Une fois tous les fichiers .o produits, ils peuvent être linkés à l'aide de `gcc -o exec_name source1.o source2.o ... sourcen.o` (`exec_name` étant le nom choisi pour le fichier exécutable).

On ne peut vraiment utiliser avantageusement la compilation séparée qu'à l'aide des makefiles couverts infra.

16.5 Les conventions de nommage

Le nommage des fonctions, variables, constantes et types doit suivre une convention précise. Cette convention doit permettre, à partir d'un nom, d'identifier le type de l'identificateur (fonction, variable, ...) ainsi que le scope s'il s'agit d'une variable. La convention suivante est proposée :

Les fonctions : des lettres minuscules. Le premier mot commence par une minuscule et chaque nouveau mot est marqué par une majuscule. Exemple : pushOnStack.

Les variables globales : des lettres minuscules. Chaque nouveau mot commence par un majuscule et les mots sont séparés par des underscores ("_"). Ex : Global_Stack.

Les variables locales : des lettres minuscules. Chaque nouveau mot commence par une minuscule et les mots sont séparés par des underscores. Ex : local_stack.

Les constantes : des lettres majuscules. Les mots sont séparés par des underscores. Ex : STACK_SIZE.

Les types : des lettres minuscules. La première lettre de chaque mot est une majuscule. Ex : StackOfInt.

16.6 Le debugging

Que faire lorsque l'on ne dispose pas de debugger symbolique (ce qui peut arriver plus souvent qu'on ne le pense) ?

La technique universelle consiste à truffer le code de `printf` permettant de suivre l'évolution du programme. Si elle est utilisée telle quelle, cette technique pose un problème. En effet, une fois le module debuggé, les *printf* seront plus intempestifs qu'autre chose (sans compter la perte de performance). Il faudra donc les enlever. Cependant, s'il est nécessaire de debugger une nouvelle fois le module, il sera alors nécessaire de replacer les `printf`. Tout ceci est très fastidieux. La solution, consiste à entourer chaque affichage de debugging des directives suivantes (l'exemple est donné pour le module `stacks.c`) :

```
#ifdef __STACKS_DEBUG
```

```
  printf( ....

#endif
```

Lorsque le module sera compilé, il suffira de définir le symbole `__STACKS_DEBUG` pour compiler également les `printf` de debugging. Définir ce symbole se fait, soit en ajoutant un `#define __STACKS_DEBUG` en début du fichier `stacks.c`, soit en ajoutant `-D__STACKS_DEBUG` à la ligne de commande du compilateur.

Il existe également une macro en C qui permet de vérifier que des invariants sont respectés. Elle s'appelle `assert` et prend comme paramètre une expression booléenne. Si cette expression est évaluée à faux, une erreur est déclenchée *au run-time*. A titre d'exemple, supposons qu'il existe une relation entre deux variables telle que la somme de ces variables doit toujours être égale à 5. A chaque exécution de `assert(x+y == 5)`, le run-time vérifiera la contrainte. Remarquons que l'utilisation intensive de `assert` ralenti l'exécution du code. C'est pourquoi, lorsque la constante `NDEBUG` est définie, toutes les macros `assert` sont ignorées.

16.7 Les bugs classiques

Voici une liste des bugs classiques. Il faut donc être très prudent lorsque l'on se place dans un contexte susceptible de générer un de ces bugs.

16.7.1 Dépassement de la taille d'un string

Cela consiste, pour un tableau ou pour une chaîne de caractères, à écrire de l'information au delà de la taille permise. Les conséquences sont catastrophiques mais, malheureusement, latentes.

16.7.2 Ecrasement de la fin de string

Cela consiste à écraser le 0x0h de fin de string. Le comportement des primitives de traitement de strings (ex : `strcpy`) est désastreux ou très surprenant.

16.7.3 Allocation des pointeurs vers des pointeurs

Supposons un vecteur de vecteurs d'entiers qu'il faut allouer dynamiquement car les tailles ne sont connues que lors de l'exécution. On donc le code suivant :

```
/* n et m sont les tailles respectives du vecteur et des vecteurs */
int **vecteur_de_vecteur;

vecteur_de_vecteur=
```

```
   (int **)malloc(sizeof(int *) * n);

for(int i=0; i<n ; i++)
{
    vecteur_de_vecteur[i]=
       (int *)malloc(sizeof(int) * m);
}
```

il est courant que la première allocation soit oubliée. Le comportement de l'intérieur de la boucle, plus particulièrement l'instruction `vecteur_de_vecteur[i]` génère alors des surprises.

16.7.4 Désallocation manquante

Les oublis de désallocation sont également courants. Dans ce cas il n'y a pas de corruption de la mémoire. Cependant, pour peu que la mémoire non libérée occupe une place non négligeable et que cela se produise plusieurs fois, il est possible de tomber à cours de mémoire.

16.7.5 Désallocation multiple

Cette erreur est le dual de la précédente. Elle consiste à désallouer plusieurs fois le même morceau de mémoire. Cela résulte généralement en une corruption du heap, dont les effets sont à retard.

16.7.6 Passage par référence miraculeux

Il s'agit dans ce cas d'une mauvais compréhension de la portée du passage par référence. Le code suivant :

```
fooBar(int *a, int *b)
{
  *a=*a+1;
  b=a;
}


barFoo(void}
{
  int x,y;

  x=5;
  y=10;

  fooBar(&x,&y);
```

```
  print("x:%d y:%d\n",x,y);
}
```

affichera `x :6 y :10` à l'appel de `barFoo`, et non `x :6 y :6` comme on pourrait éventuellement l'imaginer.

16.8 Les précautions d'usage

Concernant la prévention directe des bugs :

1. Vérifier plusieurs fois le code sensible.
2. Compiler avec l'option `-Wall` (en gcc) afin de placer le compilateur en mode paranoïaque.
3. Utiliser des librairies comme Purify (sur Irix) ou Electric Fence (sur Linux).
4. Mettre des gardes en mode debug.
5. Placer les fonctions `malloc` et `free` dans wrappers qui effectuent le comptage des appels (les compteurs doivent s'équilibrer à un moment ou à un autre dans la vie du programme).
6. Ecrire ses propres routines d'allocation et de libération et, pourquoi pas, un garbage collector.

Concernant le style d'écriture général :

1. Eviter la programmation cryptique.
2. Utiliser des conventions de nommage et des noms significatifs.
3. Ne pas "jouer à l'optimisateur" et dès lors, écrire du code illisible. Le compilateur est le principal optimisateur.
4. Ecrire des modules de taille raisonnable. Ce genre de contrainte est assurée si le design est bien fait.
5. Adopter des conventions pour les allocations et libérations de structures : qui alloue ? qui libère ?

16.9 Les outils de développement

La programmation ne se conçoit pas sans l'utilisation d'outils adéquats qui permettent, d'une part d'alléger la tâche du programmeur, d'autre part de minimiser le risque d'erreurs.

16.9.1 Les éditeurs smart

Ils offrent des possibilités telles que l'indentation automatique, la mise en couleurs des mots clés et le balancement des délimiteurs qui vont par paire (ex : paranthèses, guillemets etc ...). Exemples de tels éditeurs : jed, emacs, Xemacs etc ...

16.9.2 Make et les makefiles

Make répond au problème suivant. Si on travaille avec un grand nombre de sources, il ne faut certainement pas tout recompiler lors de chaque modification. Make identifie les fichiers modifiés et ne recompile que ceux-là ainsi que ceux qui dépendent des fichiers modifiés.

Les relations entre les différentes sources sont explicitées dans un makefile (dont le nom est `Makefile`. Les commandes permettant la compilation de chaque module sont également présentes dans le Makefile. Pour compiler, il faut taper `make` qui interprète le makefile et re-compile ce qui est nécessaire.

16.9.3 lint & cb

lint est un vérificateur syntaxique et sémantique de code C. Il existe à présent des versions ANSI C de Lint ce qui rend Lint plus utile qu'auparavant.

cb (C Beautifier) est un programme réalisant l'indentation des programmes C (l'indentation K&R est utilisée, ce qui est discutable).

```
CFLAGS = -ansi
main: test

test:  mv.o test.o
       gcc mv.o test.o -o test

mv.o:  mv.h mv.c
       gcc $(CFLAGS) -c mv.o -o mv.o

test.o: test.c
        gcc $(CFLAGS) -c test.c -o test.o

clean:  rm -rf *.o
```

16.9.4 RCS

RCS (Revision Control System) est particulièrement utile lorsqu'il s'agit de travailler à plusieurs personnes sur un ensemble de sources. RCS gère l'accès simultané aux sources ainsi que le versioning.

Pour utiliser RCS, il faut tout d'abord créer un répertoire RCS dans le répertoire des sources que l'on désire gérer. Il existe deux commandes de base : le check-in (`ci`) qui consiste à introduire (ou rendre) une source à RCS et le check-out (`co`) qui est l'opération inverse.

Pour donner le contrôle d'une source à RCS, on utilise la commande `ci filename`. RCS va alors *déplacer* le fichier vers le répertoire RCS.

Pour demander une source *pour consultation*, on utilise la commande `co filename` et RCS fournit la source avec les permissions placées en read-only.

Pour demander une source *pour modification*, on utilise la commande `co -l filename`. Dans ce cas, un seul utilisateur peut demander l'accès en mode modification.

Pour restituer une source, on utilise la commande `ci filename`.

RCS gérant le versioning, il est possible de demander une version précise (`co -rx.y filename`). Il est également possible de spécifier un numéro de version lorsque l'on retourne un fichier (`ci -rx.y filename`). Par défaut, RCS incrémente le numéro de version mineur (après le point) lors de chaque modification.

RCS maintient un log des modifications. Ce log est disponible à l'aide de la commande `rlog filename`.

Bibliographie

[UnixProgramming*] W. R. Stevens. *Advanced programming in the Unix environment.* Addison Wesley.

[UnixKernel] M. J. Bach. *The design of the Unix operating system.* Prentice Hall.

[TheCReference*] B. W. Kernighan, D. M. Ritchie. *The C programming language, 2^{nd} edition.* Prentice Hall.

[StructuredAnalysis] E. Yourdon. *Modern structured analysis.* Prentice Hall

[PowerTools*] J. Peek, T. O'Reilly, M. Loukides. *Unix power tools.* O'Reilly and Associates.

[UnixAdministration*] A. Frisch. *Essential system administration.* O'Reilly.

[ThePerlReference*] L. Wall, R. L. Schwartz. *Programming Perl.* O'Reilly.

[InfoGestion] L. Golvers. *Syllabus d'informatique de gestion.* Section Informatique et Sciences Humaines - Faculté des Sciences Politiques, Economiques et Sociales - Université Libre de Bruxelles.

[TCPIP1] W. R. Stevens. *TCP/IP illustrated, volume I, the protocols.* Addison Wesley.

[TCPIP2] W. R. Stevens, G.R. Wright. *TCP/IP illustrated, volume 2, the implementation.* Addison Wesley.

[NetProgramming*] W. R. Stevens. *Unix network programming.* Prentice Hall.

[GuideUnix*] D. Hahn. *Guide Unix de l'étudiant* . Dunod.

[RunningLinux] . M. Welsh, L. Kaufman. *Running Linux.* O'Reilly and associates.