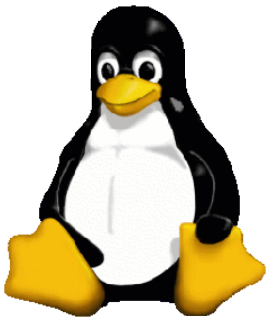
IENAC 2ème année - Majeure SITA

Programmation Système UNIX



Travaux dirigés

Joëlle Luter
2015

Sommaire

A lire impérativement avant de commencer les TD…

Le **préambule**, qui explique les objectifs généraux de ces TD et qui précise le travail demandé pour chaque TD

Les **règles de programmation en C**, qui vous éviteront beaucoup de problème pendant le codage.

La présentation de l'**environnement de développement**, qui vous présente les éditeurs de texte disponibles et le compilateur à utiliser.

Préambule

Objectifs

L'objectif de ces TD est de comprendre les mécanismes d'exécution multitâche et d'avoir un aperçu des services offerts par les systèmes d'exploitation pour les mettre en oeuvre.

L'application de ces mécanismes s'effectuera sous le système d'exploitation Linux.

Ces TD seront aussi l'occasion de consolider vos notions de programmation en C, langage que vous avez déjà abordé et qui est un des plus utilisés actuellement, surtout en informatique technique (système, réseau, informatique embarquée …).
Des règles d'écriture des programmes vous sont fournies (voire imposées…). Elles vous permettront d'écrire des programmes plus faciles à relire et plus fiables.

Assurez-vous avant de commencer à coder que chaque notion abordée est claire dans votre esprit, et n'utilisez les fonctions offertes par le système qu'après avoir compris exactement leur rôle !!!

Travail demandé

Pour chaque sujet de TD, vous rendrez :

- **un schéma d'architecture de votre application**, présentant les différentes tâches et les moyens de communication et de synchronisation mis en œuvre *(uniquement pour les programmes multitâches)*,
- **le code** abondamment commenté,
- **la liste des tests réalisés** (à mettre en commentaire à la fin du code).

Remarque pratique

Vous trouverez toutes les informations techniques nécessaires dans le cours et dans les documentations mises en ligne sur e-campus. Pensez surtout à l'utiliser le *man* UNIX, vous y trouverez le descriptif détaillé de chaque fonction, correspondant à votre version de système.

Quelques règles de programmation en C

Règles d'écriture

- Identificateurs :
  Noms des variables en minuscules `int i;`
  Noms des fonctions en minuscules `void* producteur(void*);`
  Noms des macros (*#define)* en majuscules `#define TAILLE 100`
- Utilisez des noms de variables et de fonctions **explicites** !!!
  Exemple
  ```
  int nbElements, flag_fin;
  void lanceProduction();
  ```
- Distinguez les noms des variables globales de ceux des variables locales.
  Exemple
  `int _compteur;` variable globale
  `int compteur;` variable locale
- Utilisez des macros (`#define`) et non pas des valeurs immédiates pour les constantes.
  Exemple
  Ecrivez :
  ```
  #define TAILLE 100
  ...
  int fileAtt[TAILLE];
  ```
  et non pas :
  ```
  int fileAtt[100];
  ```
- Commentez vos programmes !
- Le code doit être indenté (utilisation des tabulations).
- Une instruction par ligne !!!
- Pas de variable globale non justifiée !
- Le code retour des fonctions doit être **systématiquement** testé. En cas d'erreur effectuez le traitement approprié (en général, affichage d'un message et fin du programme).
- Rappel : le prototype de la fonction main() est `int main(void);`

Utilisation du compilateur

Lors de chaque compilation, cherchez à éliminer **toutes** les erreurs et **tous** les warnings.

Les messages du compilateur sont explicites mais en anglais. Pour les comprendre, traduisez-les mot à mot !!!

Utilisation de l'aide en ligne

L'aide en ligne (***man***) vous donnera toutes les informations concernant la syntaxe d'appel et le comportement des fonctions. Utilisez-la sans modération !!!

Résumé des principales commandes Unix

Vous trouverez dans le tableau en pages suivantes une liste des principales commandes Unix.

Vous disposez dans l'aide en ligne (commande **`man`**) du détail de chaque commande ou fonction.

Pour mémoire, le « man » est divisé en sections. Vous trouverez les sections suivantes (pour plus d'informations, tapez la commande **`man intro`**) :

- Section 1 : commandes standard Unix et commandes internes au shell
- Section 2 : Appels systèmes et Codes erreur
- Section 3 : Fonctions et bibliothèques des langages de programmation (en particulier, l'aide sur les fonctions C se trouve dans la section 3C)
- Section 4 : Formats de fichiers
- Section 5 : Divers (standards et normes, environnement, tables de caractères, …)
- Section 6 : Jeux et démos
- Section 7 : Fichiers spéciaux
- Section 9 : Drivers

La recherche d'une information par la commande man s'effectue par ordre croissant de numéro de section.

- Exemple 1 :
  La commande **`man time`** fournit l'aide sur la commande Unix `time` (section 1). Pour obtenir l'aide sur l'appel système `time()`(section 2), il faut passer la commande **`man –s2 time`**

- Exemple 2 :
  La commande **`man printf`** fournit l'aide sur la commande shell `printf` (section 1). Pour obtenir l'aide sur la fonction `C printf()`(section 3C), il faut passer la commande **`man –s3C printf`**

Chaque section dispose d'une introduction qui présente le contenu de la section et explicite certains mécanismes. Pour y accéder, tapez la commande **`man –s`***`numSection`* **`intro`**

Help!!!

man nom_commande	Aide sur une commande
man –k mot_clé	Liste des pages de man concernant le mot clé
man –sNumSection commande	Aide sur une commande de la section *NumSection*

Principales commandes relatives aux répertoires

cd [répertoire]	Changement de répertoire
pwd	Affichage du chemin absolu du répertoire courant
ls	Contenu d'un répertoire
mkdir rep1 [rep2…]	Création d'un répertoire
rmdir rep1 […]	Suppression d'un répertoire
mv rep1 rep2	Renommer un répertoire

Principales commandes relatives aux fichiers

rm fic1 [fic2 …]	Supprimer un fichier
mv fic1 fic2	renommer un fichier
mv fic1 rep1	déplacer un fichier dans un autre répertoire
cp fic1 fic2	copie avec changement de nom
cp fic1 rep1	copie dans un répertoire sans changer de nom
more fichier	Affichage d'un fichier page par page
cat fic1	Affichage d'un fichier
chmod droits fic1 [fic2…]	Modification des droits d'accès
umask	Modification du masque de création des fichiers

Principales commandes relatives aux impressions

lp,	Imprimer un fichier

Principales commandes relatives aux processus

ps	Affichage de l'activité des processus
kill [-numéro du signal] PID	Envoi d'un signal à un processus

Lancement d'un processus en background

commande [-options] [arguments] **&**

CTRL-Z.	Suspension de l'exécution d'un processus interactif
stop %numéro_job	Suspension de l'exécution d'un processus en background
jobs	Affichage de la liste des jobs
bg %numéro_job	relance l'exécution en arrière-plan d'un job suspendu
fg %numéro_job	passe en interactif un job suspendu ou s'exécutant en background

Résumé des principales commandes Unix

Redirection des entrées/sorties	**Tubes de communication (pipeline)**
commande > fic1 Redirection de la sortie standard avec écrasement commande >> fic1 Redirection de la sortie standard sans écrasement commande >**&** fic1 (csh, tcsh) Redirection des sorties avec écrasement commande 2> fic1 (sh, bash, ksh) Redirection de la sortie d'erreur avec écrasement commande >>**&** fic1 (csh, tcsh) Redirection des sorties sans écrasement commande 2>> fic1 (sh, bash, ksh) Redirection de la sortie d'erreur sans écrasement commande < fic1 Redirection de l'entrée standard	commande1 **|** commande2 [	commande3 …| commanden]

Filtres			
grep	Recherche d'une expression régulière dans un fichier	**wc**	Comptage
sort	Tri	**awk, sed**	Manipulation de fichiers texte

Expressions régulières

	Description	**Exemple**
.	un caractère quelconque	
*****	répétition du caractère suivant	*a la chaîne aa
{n**}**	répète n fois le caractère précédent	.{10} 10 car. quelconques
\	despécialise un métacaractère	* le caractère *
^	début de ligne	^a a en début de ligne
$	fin de ligne	a$ ligne terminée par a
[ ]	un des caractères entre crochets	[abc]a ou b ou c
[-]	un des caractères de l'intervalle	[a-d] a ou b ou c ou d
[^]	un caractère autre que ceux entre crochets	[^abc] tout sauf a, b et c
[^ -]	un caractère non compris dans l'intervalle	[^d-z] a ou b ou c
\x	le caractère spécial x	\n line feed

Environnement de programmation

Editeurs de texte

Plusieurs éditeurs de texte offrant des facilités spécifiques à la programmation (coloration syntaxique, numérotation des lignes,…) sont disponibles dans votre environnement.

Parmi ceux-ci, on citera :

xemacs : le plus puissant …

gedit : le plus "Windows like".

A vous de choisir, les deux remplissent parfaitement leur rôle !!!

Compilation/Edition des liens

Vous utiliserez le compilateur **gcc** (GNU C Compiler).

Commande de création d'un fichier objet (compilation seule) :

```
gcc -W -Wall -c nom_fichier_source.c
```

cette commande crée un fichier objet nommé **`nom_fichier_source.o`**

Commande de création d'un fichier exécutable (compilation + édition des liens) :

```
gcc -W -Wall -o nom_fichier_executable liste_fichiers_source
```

Explication des options :

`-W -Wall` : affichage de tous les warnings

`-o nom du fichier exécutable à créer` : permet de donner un nom spécifique à ce fichier. En l'absence de cette option, l'exécutable créé s'appellera **`a.out`**.

Commande de création d'un fichier exécutable (édition des liens seule) :

```
gcc -W -Wall -o nom_fichier_executable liste_fichiers_objet
```

Exemple de commande de compilation/édition des liens

```
gcc -W -Wall -o prog fonctions.c prog.c
```

Crée un fichier exécutable nommé **`prog`** à partir des deux fichiers source nommés **`fonctions.c`** et **`prog.c.`**

Options complémentaires

Quand vous utiliserez les threads POSIX, il sera nécessaire de spécifier des bibliothèques supplémentaires pour l'édition des liens. Vous rajouterez alors l'options **`lpthread`** .

```
gcc -W -Wall -lpthread -o prog2 prog2.c
```

Crée un fichier exécutable nommé **`prog2`** à partir du fichier source nommé **`prog.c`**. Les threads POSIX sont utilisés.

Accès fichiers, entrée et sortie standard

Cet exercice simple illustre les notions d'entrée et de sortie standard et l'utilisation des primitives d'accès fichier spécifiques au système Unix.

Réalisation d'un filtre de conversion en majuscules

Ce filtre de conversion sera réutilisé lors du TD "Transfert du contenu d'un fichier à travers des tubes".

Réalisation d'un filtre de conversion en majuscules

***Mots-clés** : langage C, descripteur de fichier, read(), write()*

Rappel

Un filtre est une commande qui prend en entrée des informations lues sur l'entrée standard (par défaut, saisies au clavier) et qui écrit ses résultats sur la sortie standard (par défaut, à l'écran). Ce type de commande doit donc fonctionner avec les mécanismes de *redirection* et de *pipeline*.

Spécifications du problème à résoudre

On souhaite réaliser un programme transformant les caractères minuscules d'un texte en caractères majuscules.

Ce programme fonctionnera comme un filtre. Les caractères minuscules seront introduits à l'aide du clavier (entrée standard) et les caractères transformés en majuscules seront affichés à l'écran (sortie standard). Les caractères non alphabétiques ne devront pas être affectés par la transformation.

Les entrées/sorties seront effectuées en accédant au clavier et à l'écran par des **appels système UNIX** et non par les fonctions de la bibliothèque C standard.

Exemple d'utilisation :

```
[luter@sundae PROG_SYST]$ ./minmaj
Bonjour
BONJOUR
azer...tyu
AZER...TYU
  nN
  NN
[luter@sundae PROG_SYST]$
```

Tests à effectuer

On vérifiera que les commandes suivantes fonctionnent correctement :

- **`./minmaj`** Affiche en majuscule chaque ligne saisie au clavier (fin de la saisie par <CTRL-D>)
- **`./minmaj < minmaj.c`** Affiche en majuscules le contenu du fichier minmaj.c
- **`./minmaj > toto`** Met les caractères saisis traduits en majuscule dans le fichier toto (fin de la saisie par <CTRL-D>)
- **`./minmaj < minmaj.c > toto`** Met dans le fichier toto le contenu traduit en majuscules du fichier minmaj.c

Informations complémentaires

On se rappellera pour effectuer la traduction que la représentation interne d'un caractère est sa valeur dans la table ASCII. Dans cette table, les codes des caractères minuscules ont une valeur supérieure au code des caractères majuscules.

En C, il est possible d'effectuer des opérations arithmétiques sur des variables de type `char`.

Exemple

```
char c='X', d;
d = c + 2;        /* d contient le code ASCII de 'Z' */
printf("---%c---\n", d);    /* affiche "---Z---" à l'écran */
```

Illustration du concept de processus

Deux TD vont illustrer la notion de processus et de communication entre processus.

Transfert du contenu d'un fichier à travers des tubes

- Création de processus
- Communication au travers du mécanisme de tube (pipeline)
- Recouvrement du code d'un processus par un nouveau fichier exécutable

Suivi du déroulement d'une application non interactive

- Communication entre processus par le mécanisme IPC de mémoire partagée

Les problèmes de synchronisation entre tâches ne seront pas évoqués dans cette partie.

Transfert du contenu d'un fichier à travers des tubes

***Mots-clés** : passage d'arguments à un programme, fichier, open(), read(), write(), close(), processus, fork(), exec(), pipe()*

Spécifications du problème à résoudre

On souhaite réaliser le traitement multiprocessus suivant :

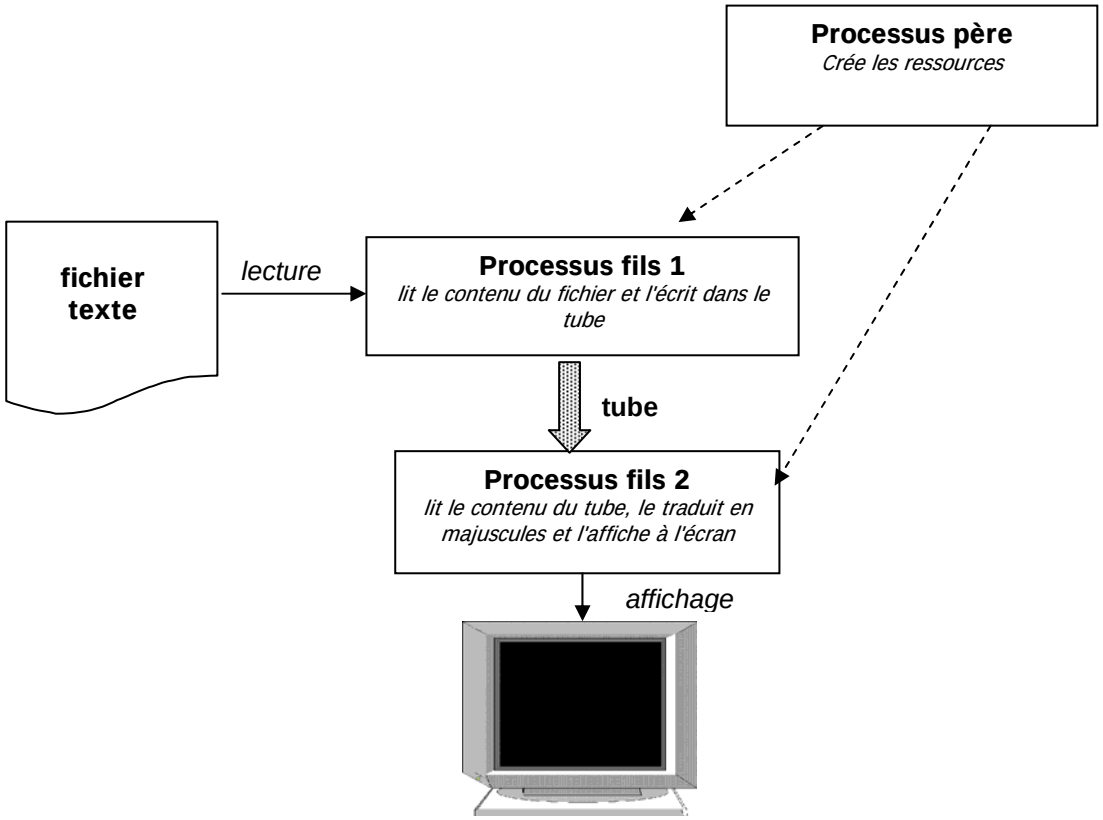


Fig. 1 Schéma de circulation des données dans l'application

Processus père

- Crée le tube (`pipe`),
- Crée 2 processus fils (`fork`),
- Attend la terminaison des processus fils (`wait`)

Processus fils 1

- Lit les données dans le fichier (`read`),
- Les écrit dans le tube (`write`).

Processus fils 2

- Lit les données dans le tube (`read`),
- Les transforme en majuscules,
- Les écrit à l'écran (`write`, sortie standard).

Indications

- Vous réaliserez le TD en 3 étapes :

  Etape 0 *(**`FicTubes1.c`**)* : Création des processus fils 1 et fils 2

  Etape 1 *(**`FicTubes1.c`**)* : La transformation en majuscules des caractères lus dans le fichier sera codée dans le programme.

  Etape 2 *(**`FicTubes2.c`**)* : La transformation en majuscules des caractères lus dans le fichier sera faite par le programme *`minmaj`* réalisé au TD1, appelé par une fonction du type *`execXX()`*.

- Un programme qui fait appel à *`fork()`* n'est pas toujours simple à relire. Pour le simplifier, écrivez des fonctions pour le code des processus fils.
- Pensez à fermer les descripteurs de tube et de fichier dès qu'ils ne sont plus utilisés, sinon le programme risque de se bloquer.

Tests à effectuer

On vérifiera que les commandes suivantes fonctionnent correctement :

**`FicTubes`*`X`* **	Affiche un message d'erreur
`FicTubes`*`X`* `minmaj.c`	Affiche le contenu du fichier `minmaj.c` en majuscules.

Vérifiez qu'à la fin de l'exécution du programme, l'invite de commande du shell s'affiche immédiatement. Si ce n'est pas le cas, remédiez à ce problème.

Suivi du déroulement d'une application non interactive

***Mots-clés** : processus, IPC, mémoire partagée, signaux*

Spécifications du problème à résoudre

Cahier des charges

On souhaite écrire un programme permettant de suivre le déroulement d'une application de calcul numérique non interactive et, si nécessaire, de terminer son exécution.

Pour faciliter ce suivi, l'application de calcul crée une zone de mémoire partagée et y écrit des informations à intervalle régulier.

Votre programme (`suivi`) sera lancé avec, en paramètre, l'action à effectuer :

`./suivi info` le programme `suivi` affiche à l'écran les informations contenues dans la zone de mémoire et se termine. Si la zone de mémoire n'existe pas, il se termine en signalant que l'application de calcul n'est pas opérationnelle.

`./suivi fin` le programme `suivi` arrête l'application de calcul, détruit la zone de mémoire partagée et se termine.

Vous pourrez télécharger l'exécutable de l'application (**`applicationTD3`**) à partir de la page *Programmation Système Unix* du site *http://rafale/mimas.*

Description de la zone de mémoire partagée

Cette zone de mémoire partagée est décrite par la structure suivante :

```
struct infoAppli {
  pid_t pid;     /* le PID de l'application */
  char message [80];  /* un texte informatif */
  float valeur;  /* le résultat intermédiaire du calcul */
};
```

La clé d'accès à cette zone de mémoire sera tirée à partir du nom du fichier exécutable de l'application : **`applicationTD3`** et de la valeur entière **`1`**.

Remarque

La fonctionnalité d'affichage des informations n'est pas considérée comme critique dans cette application. On ne jugera donc pas nécessaire de synchroniser l'accès à la zone de mémoire partagée.

Cependant, il faut être conscient que l'écriture dans la zone de mémoire partagée n'est pas une opération atomique et que, dans le cas où l'écriture et la lecture se font en concurrence, l'information récupérée par le programme de suivi peut présenter des incohérences.

Programmation concurrente

Un TD réalisé en deux étapes va illustrer la notion de programmation concurrente. La notion de tâche sera ici implémentée sous la forme de thread POSIX.

Simulation d'un parking

- Création de threads
- Synchronisation de l'exécution concurrente de 2 threads

Serveur d'affichage

- Création de threads
- Synchronisation de l'exécution concurrente de plusieurs threads
- Communication entre threads

Simulation d'un parking

***Mots-clés**: threads, sémaphores POSIX*

Spécifications du problème à résoudre

On souhaite réaliser une application **multi-threads** qui simule le fonctionnement d'un parking de 30 places. Quand le parking est plein, aucune voiture ne doit pouvoir entrer.

Un thread simulera en boucle infinie les entrées de voitures. Un autre thread simulera les sorties.

Les entrées et sorties de voitures seront affichées à l'écran. Pour faciliter le contrôle du fonctionnement de l'application, vous numéroterez les entrées et les sorties.
Dans cette étape, les threads ***entréeVoitures*** et ***sortieVoitures*** effectueront eux-mêmes les affichages.

Remarque

Les sémaphores sont utilisés ici comme compteurs de ressources.

Informations complémentaires

La fonction **`sleep()`** permet de programmer une mise en sommeil pendant une durée fixée.

Serveur d'affichage

***Mots-clés** : POSIX, threads, variables conditionnelles, file d'attente*

Spécifications du problème à résoudre

Dans le cadre d'une application multithread développée sous Unix, on souhaite réaliser un serveur d'affichage.

Une tâche écrit des messages (chaînes de caractère de longueur fixe) dans une file d'attente (tableau en mémoire). Une tâche d'affichage est chargée de les afficher à l'écran.

La tâche d'affichage doit pouvoir être suspendue et reprise sur demande par une tâche de supervision.

Gestion de la file d'attente

Il faut donc mettre en place un mécanisme de communication par file d'attente.

Le problème est un schéma producteur/consommateur où le threads *Ecriture* produit des messages, et où le thread *Affichage* les consomme.

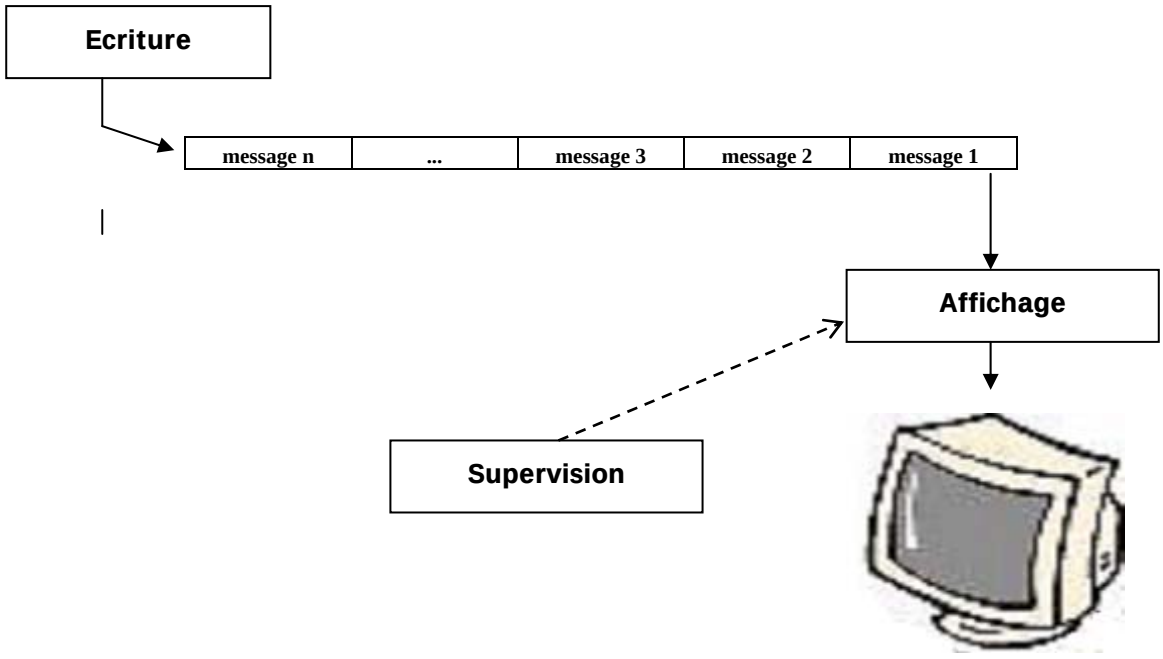
On propose d'implémenter une file d'attente en mémoire de 50 entrées de type chaînes de caractères. Cette file d'attente sera gérée comme un buffer tournant.

Les contraintes sont les suivantes:

- les messages sont consommés dans l'ordre de production,
- si la file d'attente est pleine, le producteur est mis en attente,
- si la file d'attente est vide, le consommateur est mis en attente.

Supervision

La tâche de supervision doit permettre la suspension et la reprise de la tâche d'affichage ainsi que la terminaison de l'application. Attention, elle partage l'écran avec la tâche d'affichage :!



Réalisation

Vous réaliserez le TD en 2 étapes :

Etape 1 : Pas de supervision, les tâches d'écriture et d'affichage tournent en boucle infinie et ne peuvent pas être suspendues.

Etape 2 : Mise en place de la supervision, les tâches d'écriture et d'affichage tournent jusqu'à la demande de terminaison de l'application. La tâche d'affichage peut être suspendue et reprise.