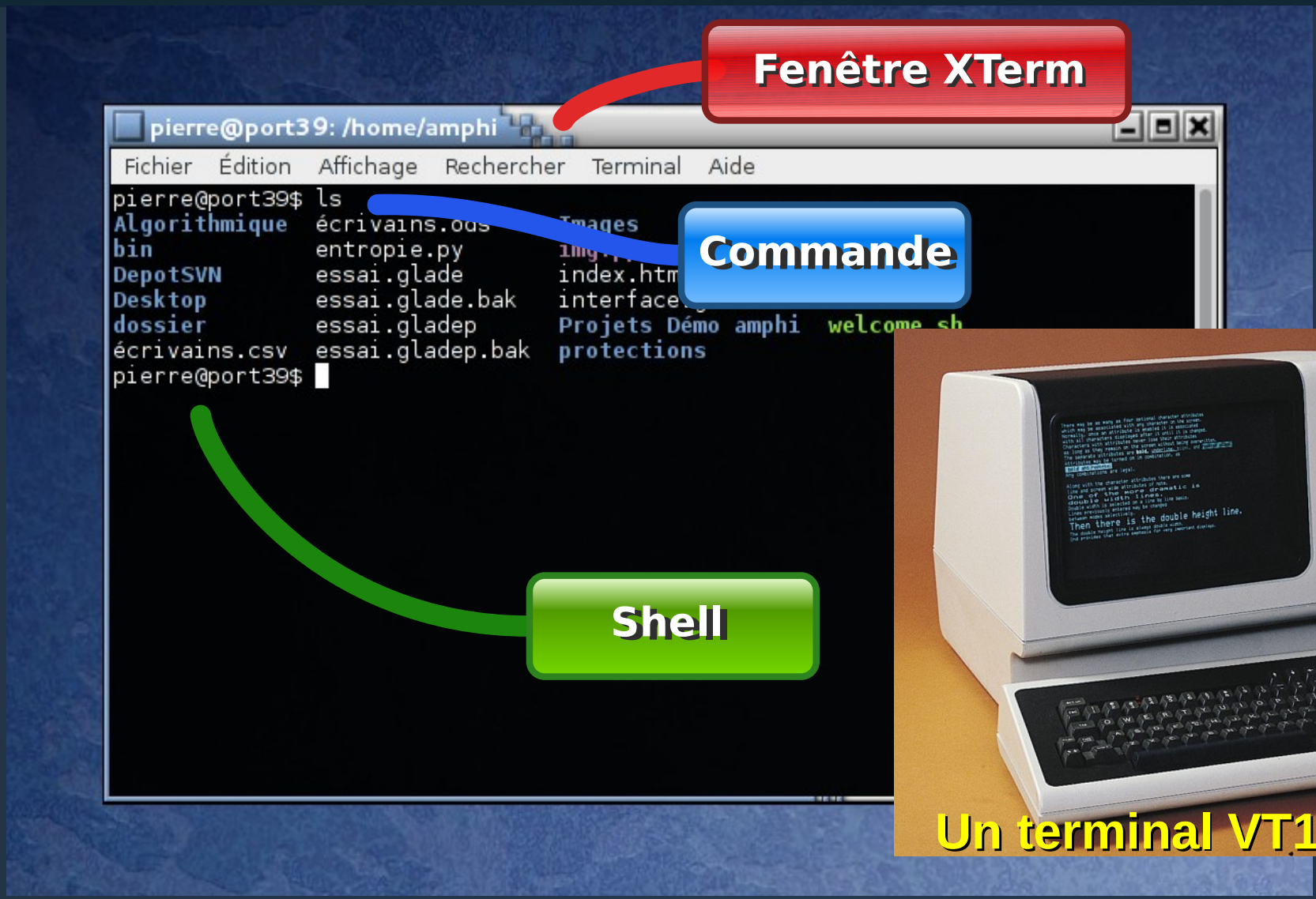
Chapitre 3 : Commandes Unix

Shell, commandes et processus

3.1 – Le Shell

« shell » = logiciel interface utilisateur pour lancer des commandes

Interface utilisateur « ligne »



Le shell

- **L’interface utilisateur s'appelle « shell »**
  - XTerm = fenêtre qui contient un shell
  - Cette fenêtre affiche seulement du texte et gère le clavier, comme les anciens VT100
  - Votre shell s'appelle **bash** (il en existe d'autres)
  - Un shell est un programme qui répète indéfiniment :
    - afficher le « prompt »
    - lire une ligne = une commande avec ses paramètres
    - exécuter cette commande

Son algorithme

- Pour les curieux, voici une idée de son algo :

```
char ligne[80];          // stockage pour une ligne
int main() {
    while (1) {          // boucle infinie
        printf("> ");    // afficher le prompt
        gets(ligne);     // lire une ligne
        system(ligne);   // exécuter cette ligne
    }
}
```

NB : c'est seulement un squelette, et il est partiellement incorrect !
(pas de test de fin et `system(ligne)`, en fait, lance un autre shell)

Lecture et édition d'une ligne

- Lors de la lecture d'une ligne, on peut :
  - Utiliser les flèches ▲ et ▼ pour réafficher une commande précédente, on peut ensuite l'éditer
  - Utiliser la touche TAB pour compléter une saisie
    - Nom de commande, nom de fichier, paramètre...
  - Utiliser le copier (sélection) coller (bouton du milieu)
  - Utiliser la notation `!initiales` pour refaire la commande qui commençait par ces initiales

Ce que le shell fait ensuite

- Après avoir lu la ligne de commande, le shell
  - Remplace les jokers * ? [ ] { } par la liste des noms de fichiers qui correspondent
  - Remplace les variables par leurs valeurs (voir la programmation en période P2).

    ⇒ Ça donne une ligne prête à être exécutée : nom de la commande, paramètres sans jokers
- Il lance l'exécution de la commande puis attend qu'elle se finisse

Exécution synchrone

- Ce mode de fonctionnement est nommé ***synchrone*** :
  - Le shell attend que la commande soit finie
  - On ne peut rien faire d'autre tant que la commande est en cours d'exécution
- Arrêt avant la fin :
  - on peut arrêter l'exécution par **CTRL-C** (noté ^C)
  - ou **en tuant son *processus***

3.2 – Processus

« processus » = commande en cours d'exécution

Qu'est-ce qu'une commande ?

- Dans le système Unix, de nombreuses **commandes** sont en réalité des **programmes compilés** (issus de sources C)
  - Les commandes de base sont dans **/bin**
  - Les commandes supplémentaires dans **/usr/bin**
  - Les commandes admin. dans **/sbin** et **/usr/sbin**
  - Les commandes perso dans **~/bin**
- Lancer une commande = exécuter un programme
- Seules quelques commandes sont traitées en interne par le shell : cd, pwd, exit…

Processus

- Un **processus** est une **commande en cours d'exécution**
  - Un système multitâche gère de nombreux processus simultanément (exécution parallèle)
- La commande **ps** donne la liste des processus :
  - ps — liste simple de mes processus
  - ps -fu *utilisateur* — liste détaillée de cet utilisateur
  - ps -edf — liste très détaillée de tous les procs

Informations fournies par ps -f

- Un processus est décrit par :
  - un **numéro unique** appelé **PID** (identifiant)
  - UID : nom de l'utilisateur qui l’a lancé (propriétaire)
  - CMD : la ligne de commande qui l’a généré
  - PPID : le n° du processus shell qui l’a créé
  - TIME : sa durée d’exécution, STIME : heure début
  - C : sa charge d’exécution, c'est un indicateur sans unité (c'est pas un %), plus il est élevé, plus le processus demande de travail
  - de nombreuses autres informations sur son état interne

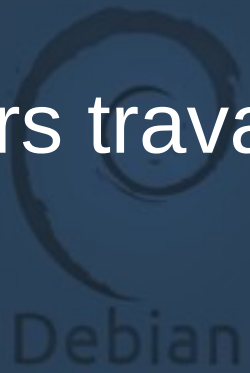
Supprimer un processus

1) **Obtenir le PID** de ce processus
   a) ouvrir une autre session : un autre xterm ou ssh
   b) utiliser `ps -fu` *`monlogin`*
   – chercher le processus dans la liste ⇒ on a son PID

2) Employer la **commande kill**
   – `kill -9` *`PID`* supprime celui qui a ce PID

3.3 – Exécution asynchrone

Pour lancer plusieurs travaux en même temps



Exécution de plusieurs programmes

- Avec une interface à fenêtres, on peut lancer plusieurs programmes sans attendre.
- Mais **avec un shell, on n'a pas la main tant que la commande en cours n’est pas finie**.
- Aucun rapport avec mono/multitâche (niveau profond dans le noyau du système) mais **mode de fonctionnement du shell** (interface) :
  - en mode texte, il est bien plus pratique de lancer les commandes une par une
  - en mode graphique, chaque commande a sa fenêtre

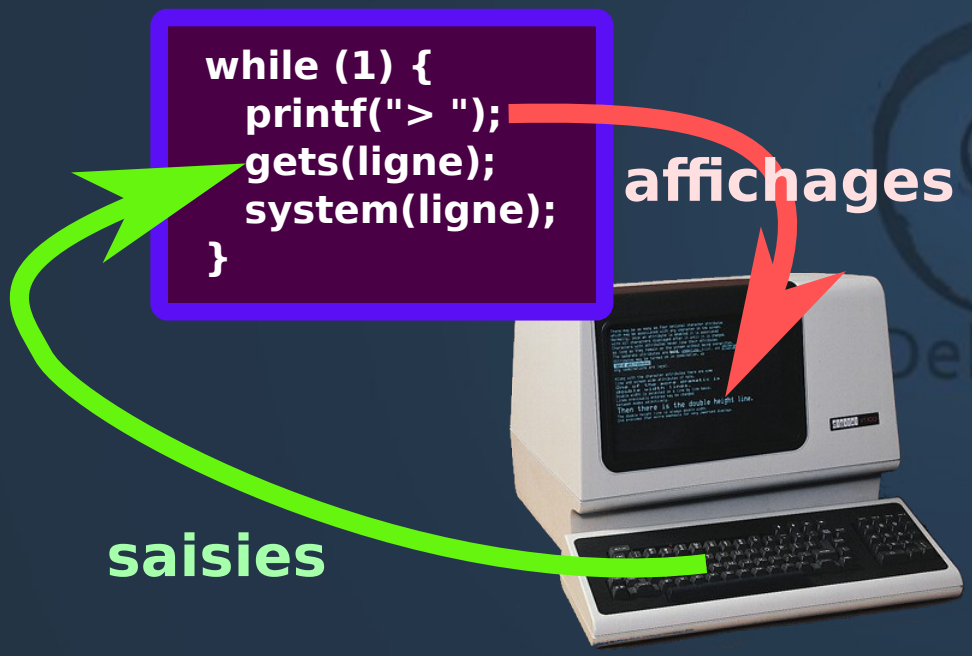
Rappel du fonctionnement

- On tape une ligne = lancement d'une commande
- Elle s'exécute ⇒ on attend tant que c'est pas fini
  - Les résultats s'affichent à l'écran
  - On peut saisir des données au clavier

    ⇒ il faut bien que le terminal soit « capturé » par la commande en cours d'exécution
- De fait, on est bloqué car il n'y a qu'un seul clavier et un seul écran

Illustration

- Shell seul

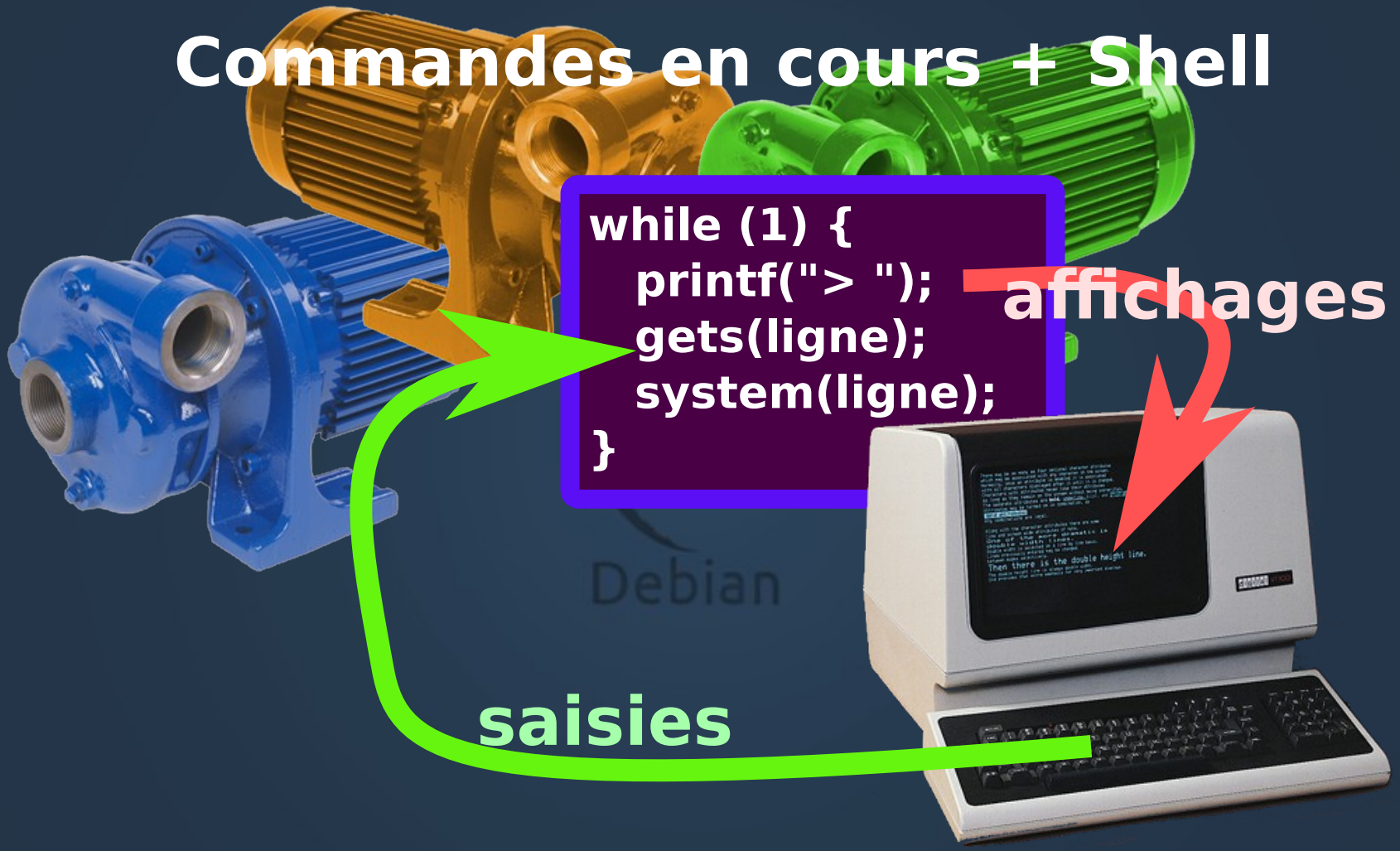


- Pendant une commande

Situation pendant une commande

- Il y a deux processus :
  - Le shell, en train d'attendre la fin de la commande
  - La commande qui travaille
- Le clavier est attribué à un seul processus
- L'écran est partagé
  - Les deux processus peuvent écrire dessus, mais ne le font pas (le shell est bien élevé)

Ce qu'on souhaiterait



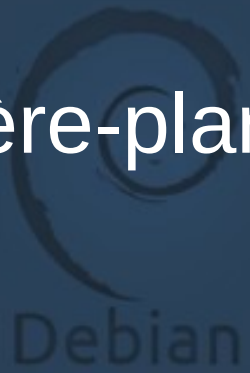
Pour exécuter plusieurs commandes en même temps

- Il suffit de lancer ces commandes en « arrière-plan » :
  - Ajouter un & à la fin de la ligne de lancement
- Exemples :
  - povray paramètres... &
  - avconv paramètres... &

  Cela lance ces commandes en tâches de fond : elles se déroulent simultanément
- De tels processus sont appelés des JOBS

3.4 – Jobs

Processus en arrière-plan gérés par le shell



Ce qui se passe

- Un processus lancé en **arrière-plan** :
  - Ne bloque pas le shell, on récupère immédiatement la main
  - on peut donc taper immédiatement une autre commande
  - Pendant ce temps, le processus s'exécute comme d'habitude
  - On peut lancer plusieurs commandes de cette manière ⇒ autant de jobs

A quoi ça sert ?

- C'est approprié pour des commandes qui fonctionnent pendant longtemps ou qui bloquent le clavier :
  - calculs long, ex : synthèse d’image, compression de données...
  - processus interactifs, ex : emacs, firefox, gimp...
  - démons et services, ex : impression, alerteur de mail...
- Inapproprié pour ls, more, rm, cc...

Limites du procédé

- Le problème, c'est qu'un processus en arrière-plan n'a pas accès au clavier (associé à la fenêtre xterm)

  ⇒ impossibilité de faire des scanf, gets et autres

- Il se retrouve immédiatement figé (en pause) s'il tente une lecture clavier

  ⇒ il faudra le débloquer pour qu'il continue

3.5 – Contrôle des jobs

Listage, arrêt, redémarrage, suppression...

Liste des jobs

- Liste des jobs existants :

  jobs

  - Affiche une liste avec un [numéro] devant chaque job
  - Indique également l'état : actif ou figé

```
[1]   En cours d'exécution   gedit arches.pov &
[2]   En cours d'exécution   povray arches.pov &
[3]+  En cours d'exécution   display abstract.jpg &
```

  - Le + indique le dernier qui a été manipulé

  – Attention le n° de job n'est pas le PID

Suppression d'un job

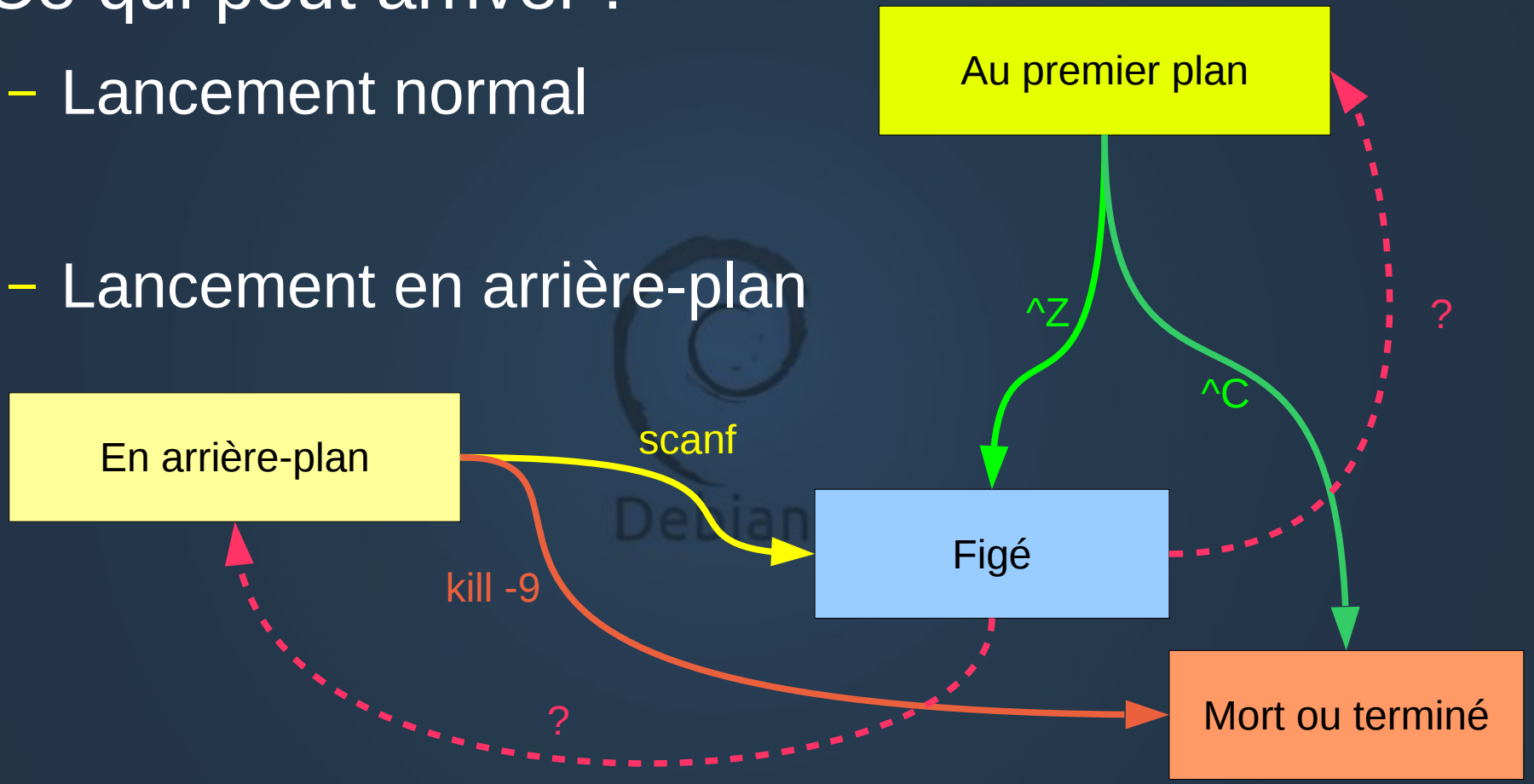
- Supprimer un ou plusieurs jobs :
  - `kill -9` les PIDs de ces jobs
  - `kill -9 %`le numéro de job
- Exemples :

```
kill -9 8768
kill -9 %2
```

- Attention au signe % s'il est présent : indique un n° de job, absent : c'est un PID

États d'un job (V1)

- Ce qui peut arriver :
  - Lancement normal
  - Lancement en arrière-plan

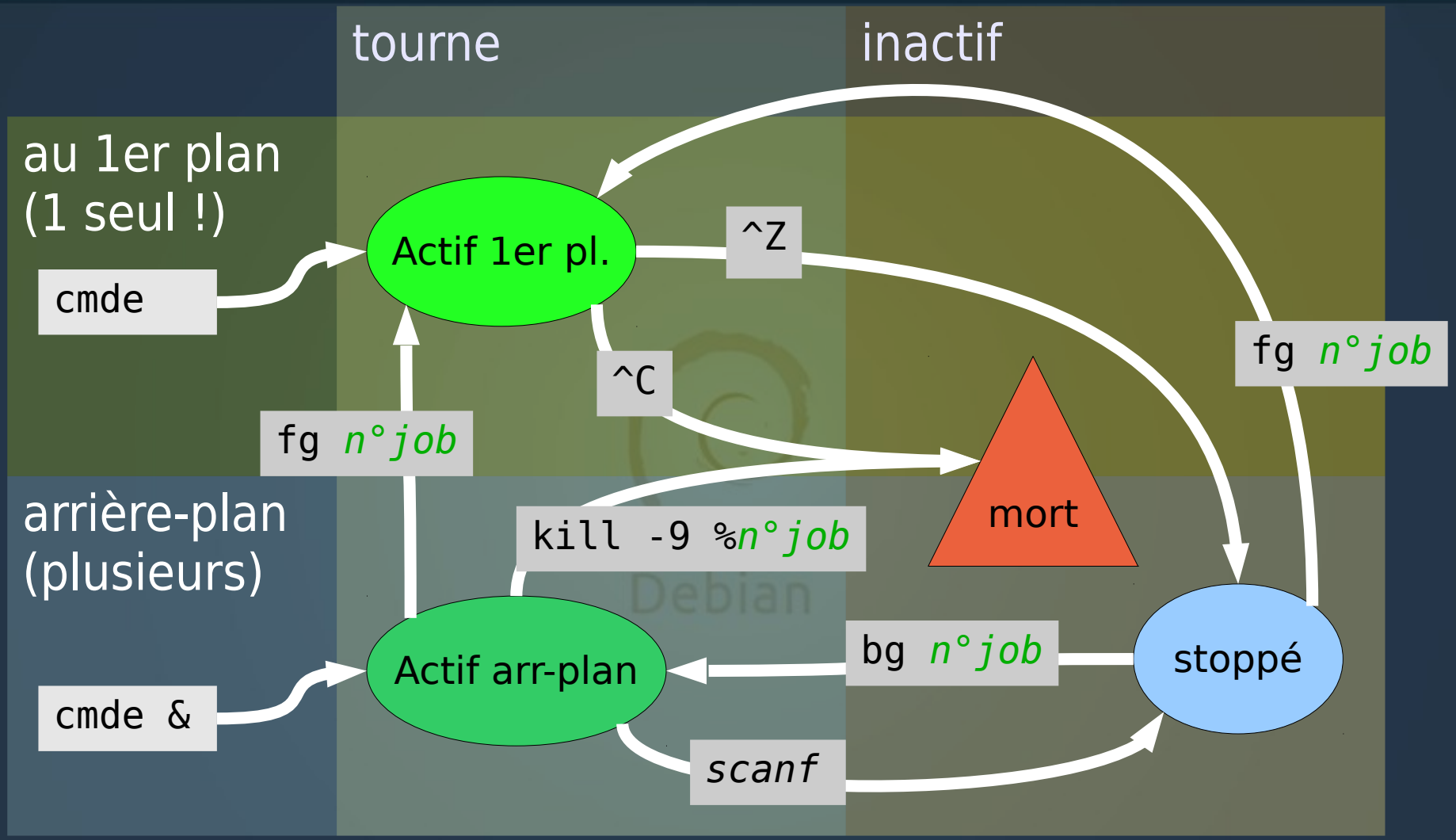


Pour relancer un job figé

- Pour le relancer
  - Au premier plan : `fg` n° de job
    - Remarque : le shell perd alors la main, c'est ce job qui accapare le clavier

      NB : le n° de job est inutile si ça concerne celui marqué +
  - En arrière-plan : `bg` n° de job
    - Le shell garde la main, le job continue en arrière-plan
- Exemples :
  - `bg 2` ⇒ le prompt revient
  - `fg 3` ⇒ on se retrouve « dans » ce processus

Diagramme des états d'un job



Séquences qu'il faut retenir

- Lancement au 1er plan et passage en arr.-plan :

```
prompt% commande
^Z
prompt% bg
```

- Lancement en arr.-plan et envoi au 1er plan :

```
prompt% commande &
```

puis, p. ex. la commande tente un scanf ⇒ figée

```
prompt % fg
```
(pour la remettre au 1er plan et faire la saisie clavier qu'elle attend)