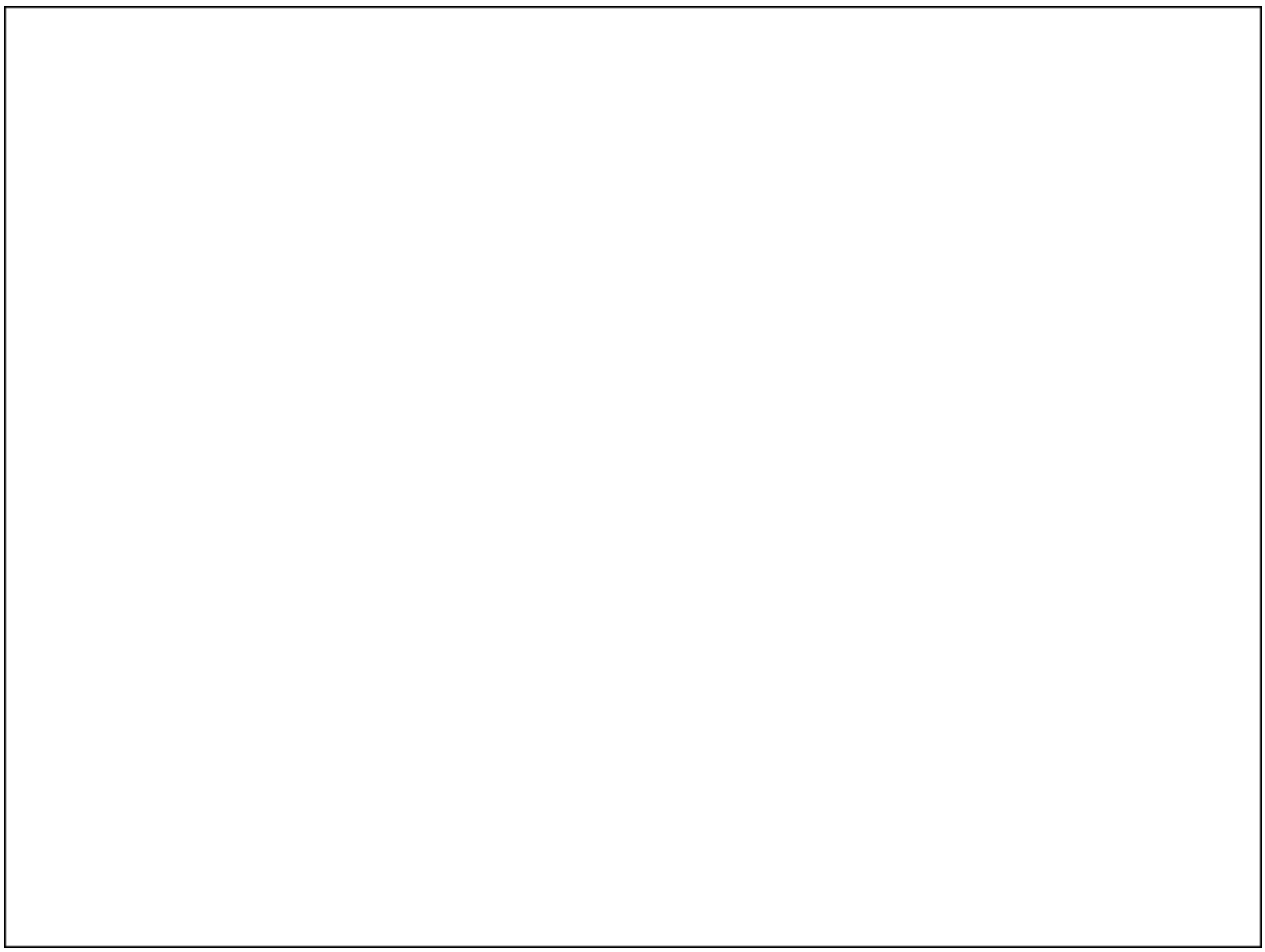
Open Source et temps réel

Optimisation d'un système Linux industriel

Christophe Blaess
http://christophe.blaess.fr

LOGILIN
http://www.logilin.fr

Séminaire Cap'tronic - 27 septembre 2013



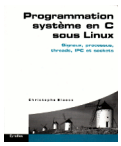
Open Source et temps réel - Optimisation d'un système Linux industriel

SOL v.1.0

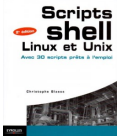
Christophe BLAESS

Ingénierie et formations Linux industriel

Programmation système sous Linux



Scripts shell sous Unix/Linux



Solutions temps réel sous Linux

Éditions Eyrolles

Axes et stratégies d'optimisations

Pourquoi optimiser un système Linux industriel ?

Gnu / Linux :

- système d'exploitation compatible Unix,
- environnement constitué de logiciels libres,
- développement essentiellement bénévole.

Linux n'a jamais été prévu pour être un système industriel, temps-réel ou embarqué !

Le cœur de cible des distributions Linux est d'abord le **serveur** (45 à 60 % marché) puis le **poste de travail** (5 % marché) mais pas les systèmes industriels.

Une optimisation est (presque) toujours nécessaire.

Que peut-on optimiser dans un système Linux industriel ?

Les demandes les plus fréquentes sont les suivantes :

- réduire la **durée de boot** (IHM pour l'embarqué) ;
- réduire l'**empreinte mémoire** du système (système embarqué fortement contraint) ;
- améliorer la qualité du **temps réel** (interactions avec des périphériques) ;
- renforcer la **robustesse** du code (drivers ou bibliothèques développés par des tiers) ;
- **accélérer** le fonctionnement global (matériel plus adapté aux demandes fonctionnelles).

Certains éléments sont antagonistes : rapidité de calcul ↔ qualité du temps réel.

Effets collatéraux positifs : empreinte mémoire → durée de boot.

L'optimisation nécessite une bonne connaissance globale du fonctionnement d'un système Gnu/Linux et une méthode de travail adaptée.

N'hésitez pas à nous interroger pour toute question concernant le développement, la mise en œuvre ou l'optimisation de vos systèmes Linux industriels : **www.logilin.fr**.

Stratégies d'optimisations

L'optimisation demande une connaissance complète de tous les éléments du système (configuration noyau, versions de bibliothèques, etc.) y compris les composants matériels essentiels (processeur, mémoire, cache, contrôleur d'interruption, périphériques, etc.).

1 – Identifier les **véritables problèmes** (goulets d'étranglement, sur-consommation, failles).

Tâche complexe, les perceptions intuitives sont rarement exactes.

Nécessite une analyse globale de tout le système puis détaillée sur chaque composant.

2 – Mettre en place une **instrumentation** donnant une quantification objective des progrès.

Facile dans certains cas (chronométrer la durée de boot).

Plus difficile dans d'autres (mesurer la consommation mémoire).

3 – Améliorer les performances par **essais successifs**.

Ne faire varier qu'un paramètre à la fois, ne pas hésiter à revenir en arrière.

Travail long et fastidieux ! Certaines parties sont automatisables (scripts).

Quelques outils de diagnostic sur l'état du système :

- **`dmesg`** : historique des messages du kernel depuis le démarrage ;

- **`ps`**, **`top`** : états et activités des processus et threads présents ;
- **`time`** : durée d'exécution d'un processus ;

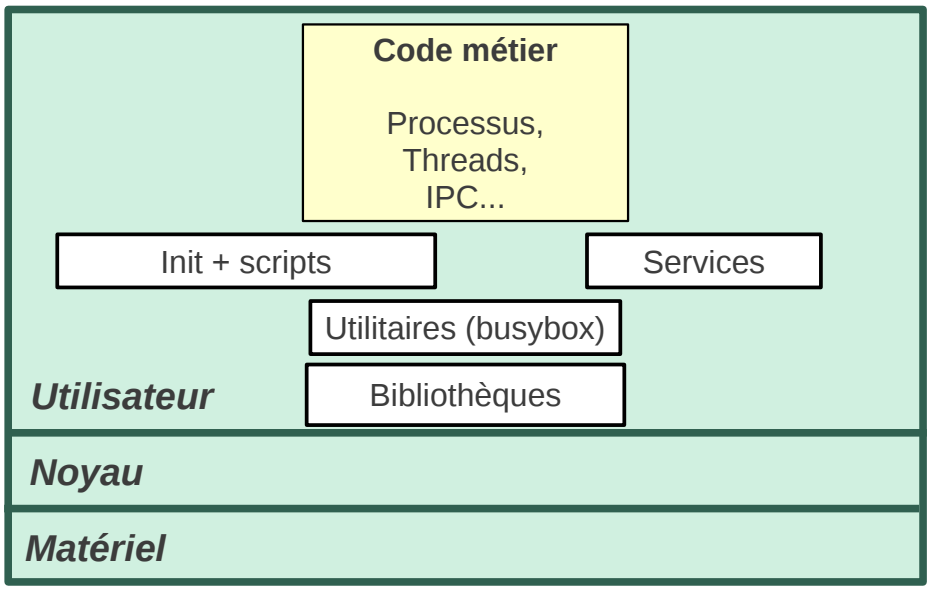
- **`free`** : aperçu de la mémoire occupée et disponible ;
- **`vmstat`** : statistiques sur l'état du système (mémoire, CPU, E/S) ;

- **`hwinfo`** : matériel présent ;
- **`/proc/cpuinfo`** : CPU(s) présent(s) ;
- **`slabinfo`**, **`/proc/vmstat`** : gestion de la mémoire dans le noyau ;

- **`latencytop`** : recherche des points de blocage de tâches en attente ;
- **`powertop`** : recherche des tâches les plus consommatrices de CPU.
- **`cyclictest`** : mesure des fluctuations de timers temps réel.
- **`oprofile`** : *profiling* intégré dans le noyau pour code utilisateur ou kernel.

De nombreux outils sous Linux se manipulent depuis la ligne de commande. Ceci permet de les invoquer facilement de manière automatisée dans des *scripts* (par exemple passage d'une batterie de tests et de mesures après une modification).

Actions sur le système



Exemple de système Linux embarqué minimal

Les différences entre un système embarqué, un poste de travail classique et un serveur reposent surtout sur les implémentations mêmes des utilitaires, des bibliothèques et des services, mais le schéma global reste identique.

Amélioration du temps de démarrage

On cherche généralement à améliorer le temps **subjectif de démarrage**, durée s'écoulant entre la mise sous tension et l'instant auquel l'utilisateur peut commencer à interagir avec le système.

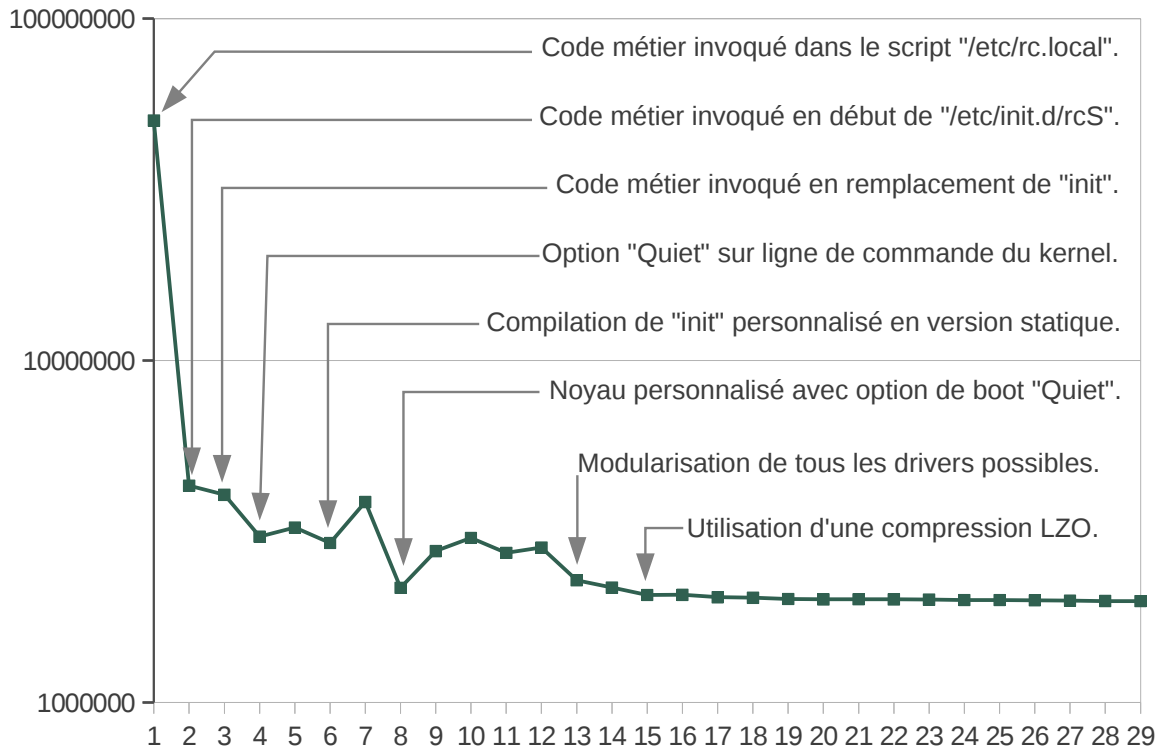
- Afficher un écran de bienvenue le plus tôt possible (animé, aléatoire, informatif, etc.)
- Optimiser le temps de chargement du système (noyau, bibliothèques...)
- Retarder certaines initialisations non indispensables immédiatement (périphériques de stockage, réseau, USB...)
- Pré-traiter les données utilisées par les applications-métier pour optimiser leur temps de démarrage.

Conseils :

- pour optimiser le temps de démarrage du système, agir d'abord sur les scripts d'initialisation et l'appel du code métier, puis affiner ensuite la compilation du noyau ;

- pour assurer la pérennité de votre système embarqué, évitez les modifications dans le code du kernel (*patches* fournis par des tiers).

Optimisation du temps de boot du système



N°	Description	Durée (µs)
1	Code métier invoqué dans le script "/etc/rc.local".	**50**312471
2	Code métier invoqué en début de "/etc/init.d/rcS".	**4**306846
3	Code métier invoqué en remplacement de "init".	**4**047723
4	Option "Quiet" sur ligne de commande du kernel.	**3**055651
5	Chargement d'un noyau non compressé (abandonné).	**3**248308
6	Compilation de "init" personnalisé en version statique.	**2**926811
7	Compilation d'un noyau personnalisé sans option "Quiet".	**3**856673
8	Noyau personnalisé avec option de boot "Quiet".	**2**164410
9	Désactivation d'options de "KERNEL_HACKING".	**2**771021
10	Désactivation du support pour "printk()".	**3**028892
11	Suppression option de boot "Quiet".	**2**736373
12	Compilation de tous les drivers nécessaires en statique.	**2**837621
13	Modularisation de tous les drivers qui l'acceptent.	**2**275120
14	Désactivation des options de mesure de performance.	**2**166897
15	Utilisation d'un algorithme de compression LZO.	**2**060790
16	Désactivation de CPU_FREQ.	**2**064312
17	Désactivation de systèmes de fichiers inutiles.	**2**030989
18	Suppression d'options réseau inutiles.	**2**023285
19	Suppression de drivers inutiles.	**2**006512
20	Modularisation du sous système SCSI.	**2**001461
21	Suppression du support pour Swap.	**2**003248
22	Diminution de la taille du buffer de log.	**2**002122
23	Désactivation du système de fichiers de debug.	**1**997615
24	Optimisation des instructions pour le processeur.	**1**992374
25	Utilisation d'un mode non "tickless".	**1**991429
26	Suppression d'ordonnancements d'I/O inutiles.	**1**989030
27	Noyau en "Voluntary preemption"	**1**984259
28	Noyau non préemptible.	**1**979666
29	Désactivation de la protection de pile.	**1**978850

Limitation de l'empreinte mémoire

Empreinte mémoire = **Taille du *bootloader***

\+ **Taille du noyau**

\+ **Taille du « *root filesystem* »**

Le noyau Linux nécessite un système de fichiers contenant scripts de démarrage, utilitaires systèmes, bibliothèques et code métier.

Ne pas confondre les empreintes :

- mémoire de masse (flash, EEPROM...) – image généralement compressée et facilement mesurable ;
- Ram allouée statiquement (code et données persistantes) – certaines commandes **`size`**, **`objdump`**, etc. donnent une indication des volumes (*bss* + *text* + *data*) ;
- Ram allouée dynamiquement (données) – mesurable uniquement par l'observation du système en fonctionnement.

La taille du noyau se réduit en limitant les options de compilation.

La plupart des problèmes liés au volume mémoire sont dus à une fuite mémoire dans le code métier !

Amélioration de la réactivité du système

On peut améliorer la réactivité d'un système à différentes stimulations :

- temps de réponse aux **interruptions matérielles** : dialogue avec des dispositifs externes, liens de communication, pilotage de processus industriels, etc.

- granularité et précision des **timers logiciels** : déclenchement d'événements cadencés régulièrement, émission de données dans des créneaux temporels précis, etc.

- **priorités des tâches** : précédence de réponse à un événement externe, réactivité de l'interface utilisateur, traitements en tâche de fond...

Le noyau Linux standard propose un ordonnancement des tâches en temps partagé (le plus équitable possible) ou en temps réel souple (*soft realtime*). On peut améliorer son comportement temps réel avec le patch **`PREEMPT_RT`**, voire approcher les performances du temps réel strict (*hard realtime*) avec Xenomai.

Noyau standard (Vanilla)

La réactivité du système aux sollicitations externes peut être améliorée en jouant sur des options de compilation du kernel. Entre autres :

- Désactiver « *Tickless System* » (**`CONFIG_NO_HZ`**)
- Activer le support « *High Resolution Timer* » (**`CONFIG_HIGH_RES_TIMERS`**)
- Activer « *Preemptible Kernel (Low-Latency Desktop)* » (**`CONFIG_PREEMPT`**)

Performances attendues

Timers :

- Période minimale : quelques dizaines de microsecondes.
- Fluctuations maximales : quelques centaines de microsecondes.

Réponse aux interruptions externes :

- Temps de réponse moyen : quelques microsecondes
- Fluctuation des temps de réponse : centaines de microsecondes.

Priorités :

- Les tâches applicatives les plus prioritaires sont néanmoins préemptées par des traitements d'interruptions éventuellement longs.

Noyau Linux PREEMPT_RT

Le *patch* Linux PREMPT_RT fourni par Ingo Molnár et Thomas Gleixner s'applique sur les sources d'un noyau standard et améliore ses performances pour les aspects liés au temps réel.

- Activation de l'option « *Fully Preemptible Kernel* » (**`CONFIG_PREEMPT_RT_FULL`**)

Après re-compilation du kernel, les performances sont améliorées instantanément sans aucune modification des applications de l'espace utilisateur.

Performances attendues

Timers :

- Période minimale : quelques dizaines de microsecondes.
- Fluctuations maximales : quelques dizaines de microsecondes.

Réponse aux interruptions externes :

- Temps de réponse moyen : quelques microsecondes
- Fluctuation des temps de réponse : dizaines de microsecondes.

Priorités :

- Certaines tâches applicatives peuvent avoir une priorité supérieure à celles des traitements d'interruptions.

Attention, l'application du patch **`PREEMPT_RT`** améliore les performances temps réel (dans les cas extrêmes) mais dégrade (très légèrement) les performances moyennes du système (communication entre les tâches, etc.).

Xenomai

Le patch Xenomai ajoute sous le noyau Linux un hyperviseur (*Adeos*) qui capture toutes les interruptions matérielles et les diffuse prioritairement à un micro-noyau temps réel strict (*Nucleus*) puis au kernel Linux standard.

Inconvénient : pour profiter des performances de Xenomai, le code applicatif doit être écrit en utilisant son interface de programmation (API) spécifique.

Performances attendues

Timers :

- Période minimale : quelques dizaines de microsecondes.
- Fluctuations maximales : quelques microsecondes.

Réponse aux interruptions externes :

- Temps de réponse moyen : quelques microsecondes
- Fluctuation des temps de réponse : quelques microsecondes.

Priorités :

- Les tâches Xenomai sont plus prioritaires que les traitements d'interruption par le noyau Linux.

Selon certaines circonstances, les tâches Xenomai peuvent se retrouver temporairement soumises aux traitements du noyau Linux. C'est ce que l'on nomme le passage en *mode secondaire*. Une part importante du travail de développement pour Xenomai consiste à éviter ou contrôler ces passages.

Amélioration du code métier

À la conception du système

Éviter les langages interprétés ou les exécutions dans des machines virtuelles : le rapport de temps d'exécution entre un code écrit en Javascript et un code en C est de 5 environ.

Dimensionner raisonnablement le processeur : le rapport de temps d'exécution d'un code de calcul entre un smartphone et une machine de bureau est de l'ordre de 10 (cf. *Geekbench*).

La conception même du code métier influe énormément sur les performances

- monotâche avec *polling* et timers,
- mutitâche en un processus multithread,
- système multi-processus avec IPC (Inter Process Communications).

Le *Garbage Collector* est à bannir pour les systèmes embarqués : ce mécanisme ne devient efficace qu'avec au moins 4 à 6 fois plus de mémoire disponible que le minimum de mémoire indispensable pour le fonctionnement.

« *Quantifying the Performance of Garbage Collection vs. Explicit Memory Management* » *Matthew Hertz & Emery D. Berger 2005* ».

Ajustement lors du prototypage du projet

Répartir explicitement les tâches et les gestionnaires d'interruption sur les cœurs disponibles.

Attention aux effets de cache des processeurs multicœurs !

Donner des priorités temps réel aux tâches (processus ou threads) en fonction de l'urgence de leur réponse aux sollicitations externes.

Débogage et optimisations finales

Pour la recherche de fuites mémoire ou de débordement de zones mémoire (*buffer overflow*) : utiliser **`valgrind`** ou **`mtrace`**. Valgrind permet également de vérifier les accès concurrents à la mémoire dans les applications multithreads.

L'optimisation d'un code applicatif commence par un travail de diagnostic et de profiling. Quelques outils : **`gcov`**, **`gprof`**, **`oprofile`**, **`LTTng`**, etc.

Conclusion et questions

L'optimisation d'un système Linux industriel fait intervenir :

- un choix judicieux des **composants du système** (bibliothèques, scripts, etc.),
- une bonne connaissance des options de **compilation du noyau**,
- une **architecture logicielle** ajustée au système et au matériel sous-jacents
- une utilisation pertinente de l'**API du système**.

Questions ?

N'hésitez pas à nous contacter sur ***www.logilin.fr***.