Du micro-contrôleur au PC embarqué

Du firmware sur micro-contrôleur à l'application embarquée

Architectures, techniques et méthodes logicielles

Christophe Blaess
http://christophe.blaess.fr
http://www.logilin.fr

Christophe BLAESS

Ingénierie et formations Linux industriel

Programmation système sous Linux

Scripts shell sous Unix/Linux

Solutions temps réel sous Linux

Éditions Eyrolles

Articles dans « Gnu/Linux Magazine France » et « Open Silicium »

Blog : christophe.blaess.fr

Architecture d'un système Linux embarqué

- Composants logiciels
- Pilote de périphériques - drivers
- Ressources nécessaires
- Temps de boot
- Services offerts par le noyau Linux

Structure d'une application Linux

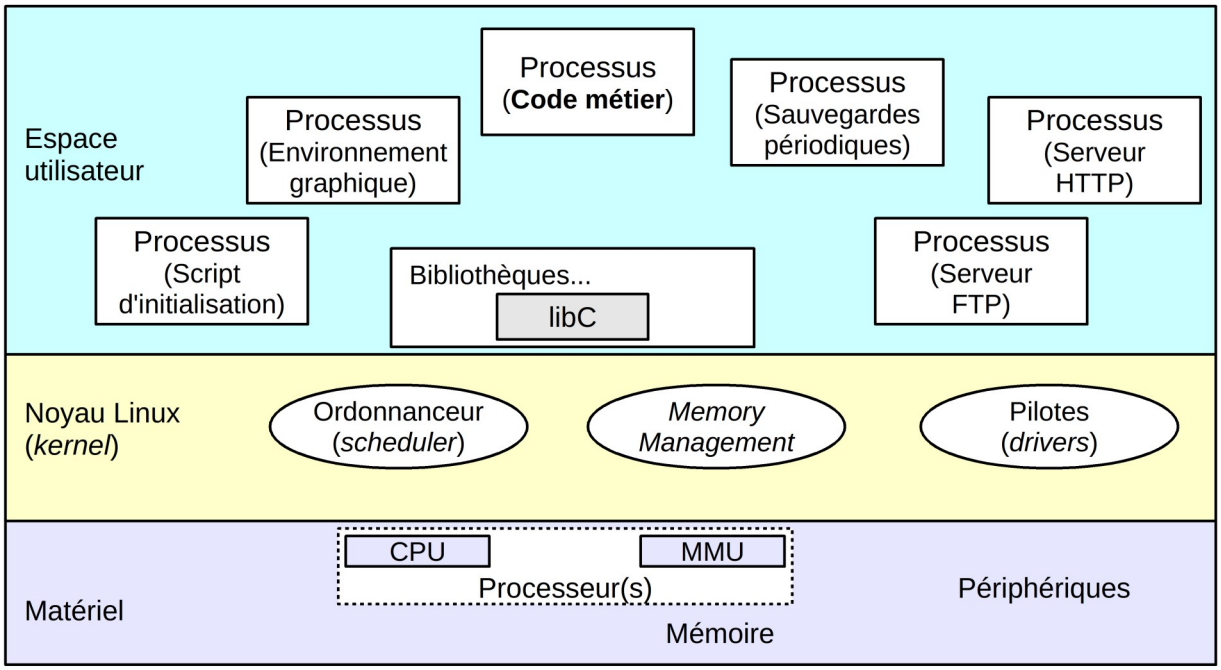
- Traitement d'événements externes
- Multitâche : processus et threads
- La mémoire virtuelle
- Communications entre processus
- Programmation multicœur et temps réel

Environnement Linux et méthodes logicielles

- Outils de développement libres
- Licences GPL et LGPL
- Conclusion et questions

Architecture d'un système Linux embarqué

Composants logiciels

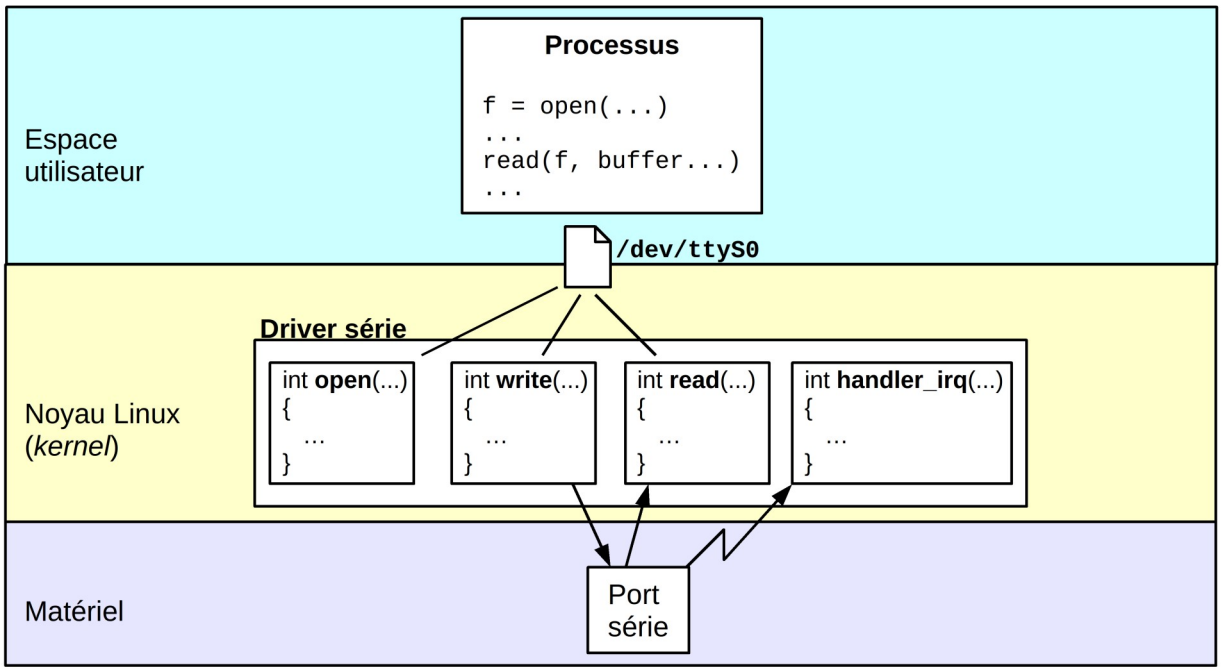


Le rôle du **noyau** Linux : mettre les ressources offertes par le **matériel** à disposition des applications de l'**espace utilisateur**.

Lorsqu'il exécute le code du noyau, le processeur est en mode **superviseur** (privilégié). Pour exécuter du code en espace utilisateur, il passe en mode protégé (non-privilégié).

Les **tâches** (*threads*) de l'espace utilisateur s'exécutent dans des **processus** (espaces de mémoire disjoints).

Pilote de périphériques - *drivers*



Pour dialoguer avec un **périphérique** (par exemple un port série), une application effectue des opérations (ouverture, lecture, écriture, paramétrage...) sur un **fichier spécial**. Le noyau convertit ces opérations en accès vers le matériel.

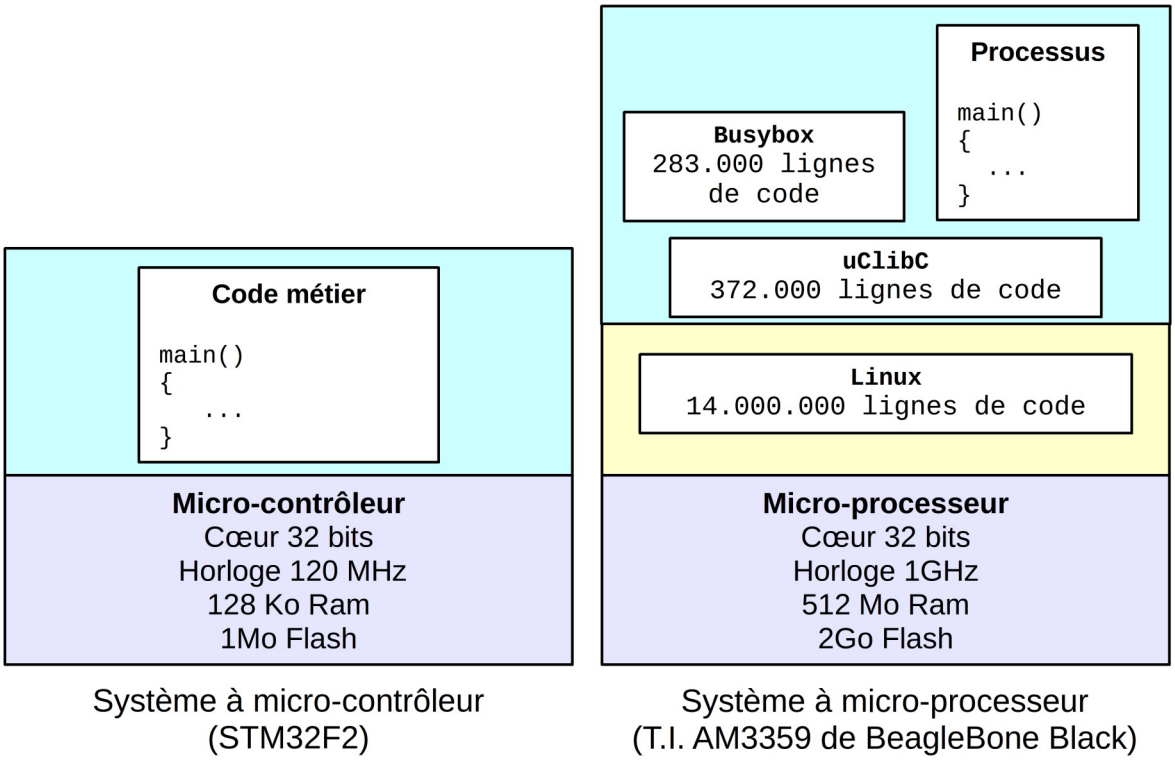
La gestion des **interruptions** est réalisée par les drivers se trouvant dans le kernel.

Pour en savoir plus

- Formation Logilin : « *Écriture de drivers et programmation noyau Linux* »

  http://www.logilin.fr/formations/

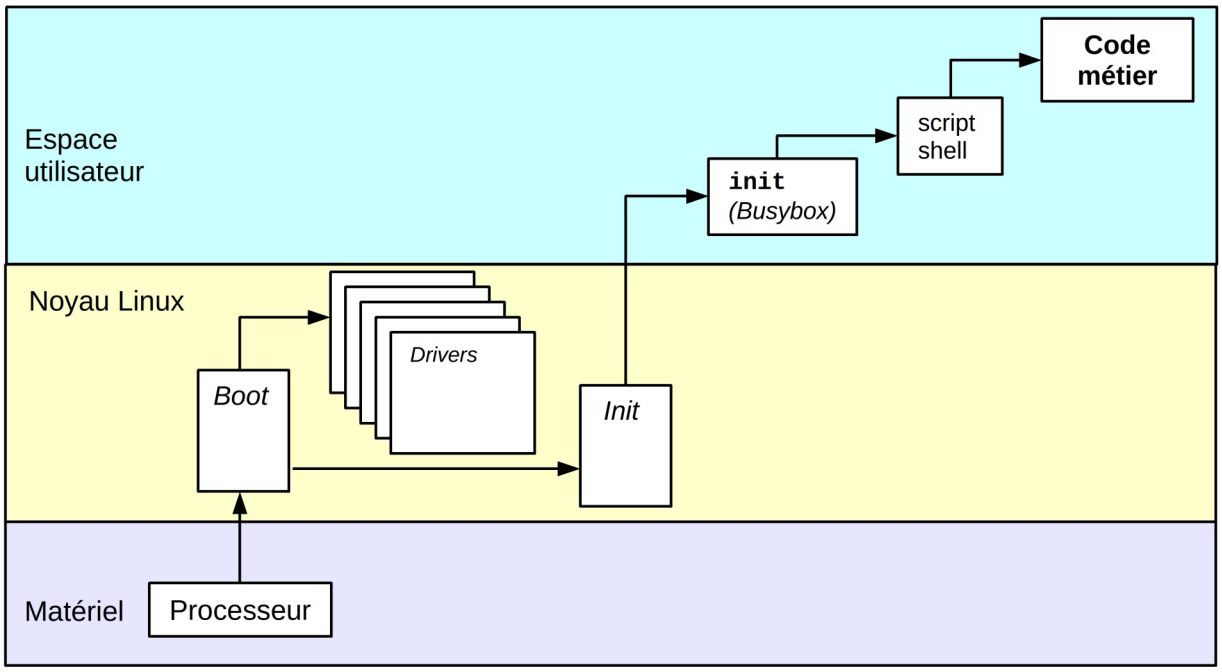
Ressources nécessaires



Sur un système à **micro-contrôleur**, le code métier est le **seul maître à bord**, il accède à volonté aux périphériques, à la mémoire, etc.

Au contraire, dans un système à **micro-processeur**, le code métier n'est qu'**une petite partie de l'ensemble** du logiciel. Il est soumis à l'ordonnancement et au sous-système de gestion mémoire du noyau. Il s'appuie sur des bibliothèques et des utilitaires externes.

Temps de boot



Suivant le type de processeur et la complexité du matériel, le temps de boot du **noyau** dure de **deux à cinq** secondes environ.

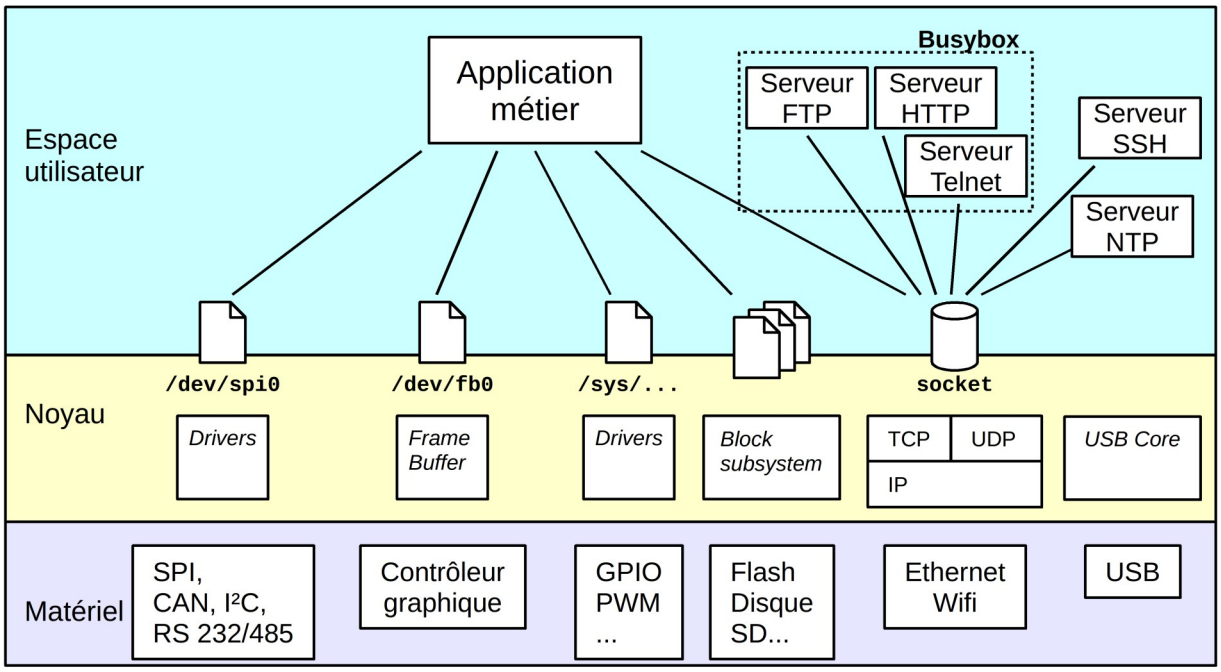
Le démarrage du processus `init`, les **tâches administratives** (montage systèmes de fichiers, configuration paramètres de `/proc`, etc.) prennent **une à deux secondes** supplémentaires.

Le lancement de tous **les services** (réseau, authentification, environnement graphique, etc.) peut demander **une dizaine de secondes**.

Pour en savoir plus

« *Optimisation du temps de boot d'un système Linux embarqué* » Christophe Blaess – Open Silicium 9, voir http://www.blaess.fr/christophe/articles/ .

Services offerts par le noyau Linux



Le noyau Linux permet directement d'accéder à plusieurs **centaines de périphériques** et **protocoles de communications**, de réaliser des **opérations d'entrées-sorties** aisément, d'afficher une **interface graphique**, de lire des dizaines de formats de **systèmes de fichiers**, de dialoguer avec de nombreux **protocoles réseau**, etc.

Si un élément n'est pas présent dans une configuration embarquée, il suffit généralement de recompiler le noyau (voire de ne compiler qu'un module) pour l'intégrer.

Structure d'une application Linux

Traitement d'événements externes

Code micro-contrôleur typique

```
int main(void)
{
    // Initialisation
    P1DIR = BIT4;  // Broche 1-4 en sortie
    ...
    // Boucle de travail
    while (1) {
          if ((P1IN & 0x20) != 0) {  // Capteur 1
                P1OUT |= 0x10;       // Actionneur Broche 1-4
                ...
          }
          if ((UART_STATUS & RX_CHAR) != 0) {  // Octet reçu
                traiter_caractere(UART_RXBUFFER);
                ...
          }
          // etc...
    }
    return 0; // Jamais atteint
}
```

Avantages :

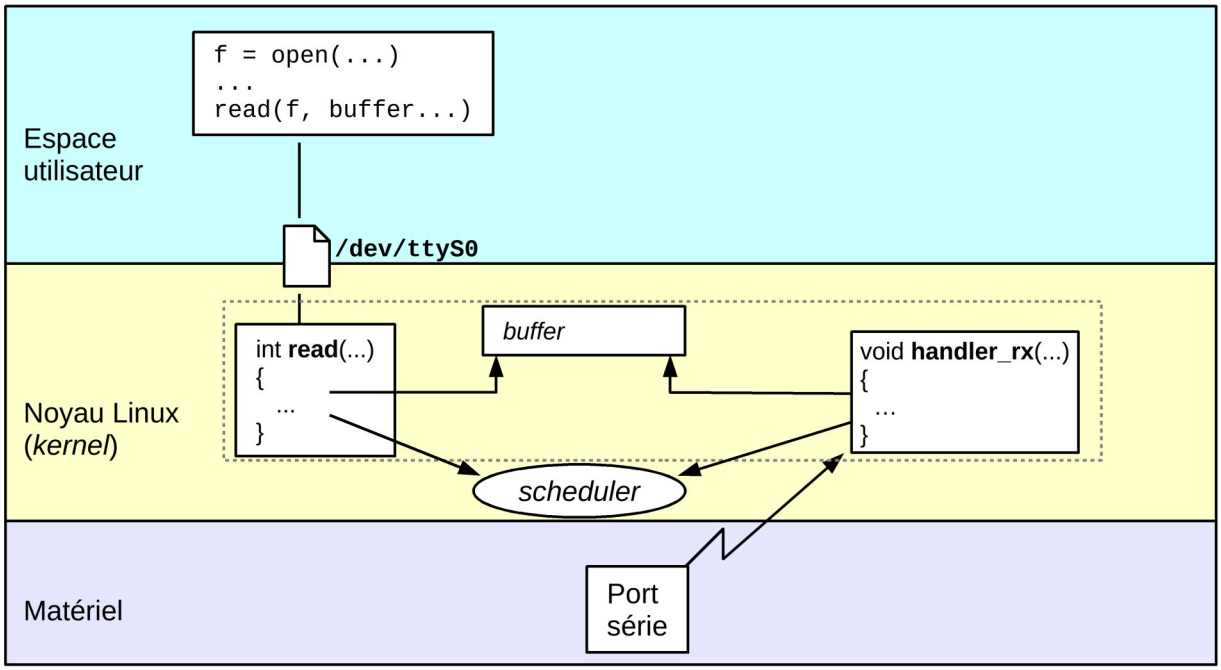
- Simplicité d'écriture initiale,
- Relative simplicité de maintenance, mais attentions aux « verrues logiques » (cas particuliers, contournements, etc.).

Inconvénients :

- Difficulté à intégrer des opérations d'attente, des timers, etc.,
- Pas de déterminisme des temps de réponse aux modifications des entrées (la durée de parcours de la boucle dépend d'autres conditions se trouvant plus bas),
- Consommation électrique importante, et élévation de température du processeur.

La programmation typique pour micro-contrôleur ne fonctionne pas correctement sur un micro-processeur avec système d'exploitation.

Mise en sommeil – Opération bloquante



Un processus utilisateur, pour recevoir des données d'un port série par exemple, effectue un **appel-système** `read()`.

La méthode `read()` du driver vérifie s'il y a des données disponibles dans son ***buffer* d'entrée**. Si c'est le cas, elle renvoie immédiatement les données au processus qui continue son déroulement.

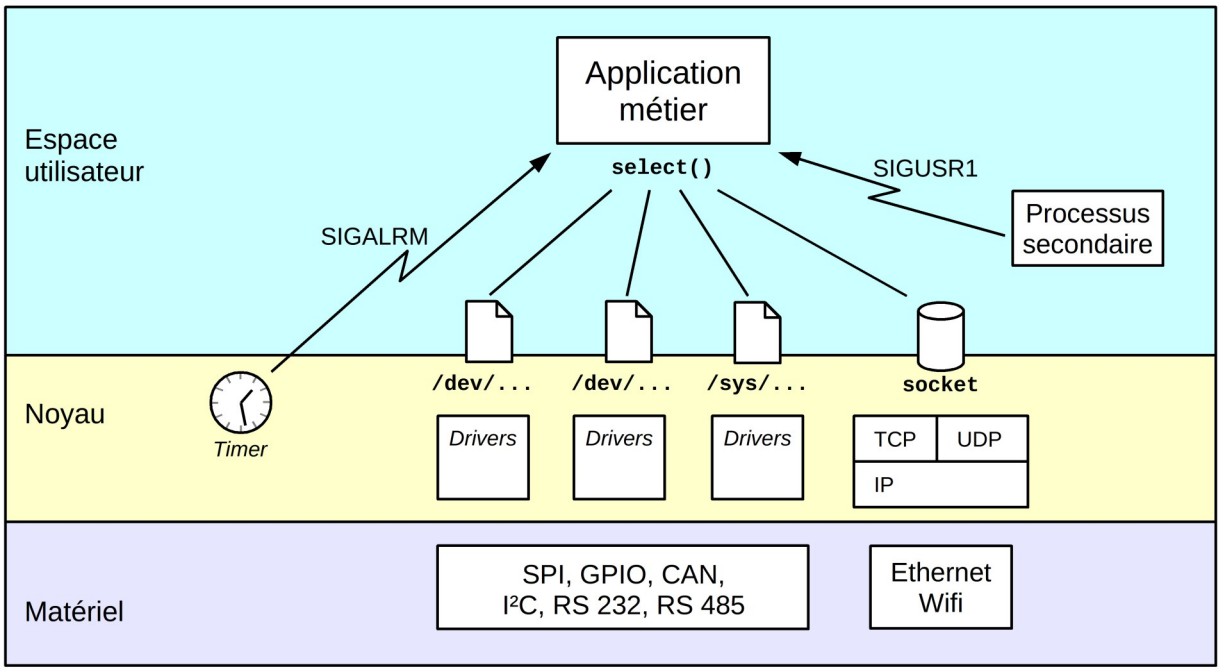
Si aucune donnée n'est présente, la méthode `read()` **endort le processus** en invoquant le *scheduler* qui pourra ainsi ordonnancer – activer – d'autres tâches.

Dès que le port série indique (par l'intermédiaire d'**une interruption**) qu'un caractère est disponible, le *handler* associé va lire les données et les placer dans le *buffer* du driver, puis demander au *scheduler* de **réveiller** le processus en attente.

Avantages :

- **Consommation électrique et échauffement très faibles** lorsqu'il n'y a pas d'activité d'entrées-sorties (s'il n'y a aucune tâche en attente, le scheduler endort le processeur en attente de l'occurrence d'une interruption).
- **Déterminisme** du temps écoulé entre réception d'une interruption et début du traitement par le code métier. À moduler toutefois en fonction de la configuration temps réel.

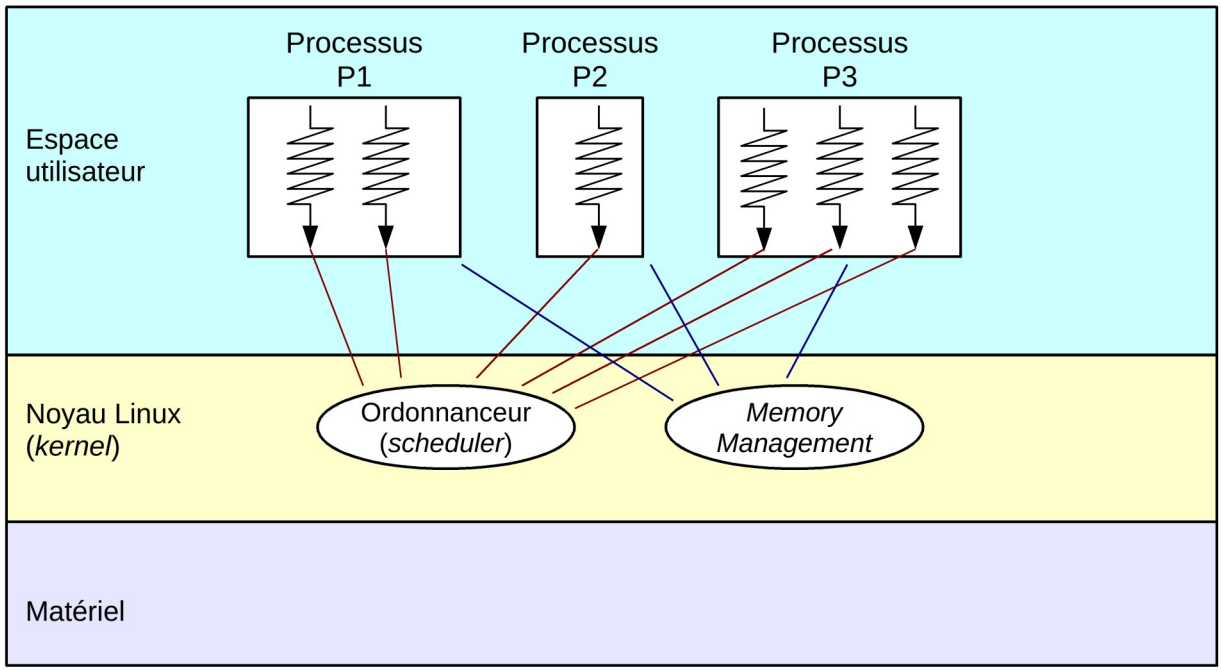
Interactions typiques d'une application Linux



Une application Linux peut réaliser des entrées-sorties efficaces en utilisant :

- l'appel-système **`select()`** pour surveiller simultanément l'état de divers périphériques et communications réseau ;
- un réveil sur *timer* programmé par l'appel-système **`setitimer()`** pour réaliser des actions périodiques ;
- une notification par signal en provenance d'autres processus lors de l'occurrence de certaines conditions, l'action à réaliser étant programmée par l'appel-système **`sigaction()`**.

Multitâche : processus et threads



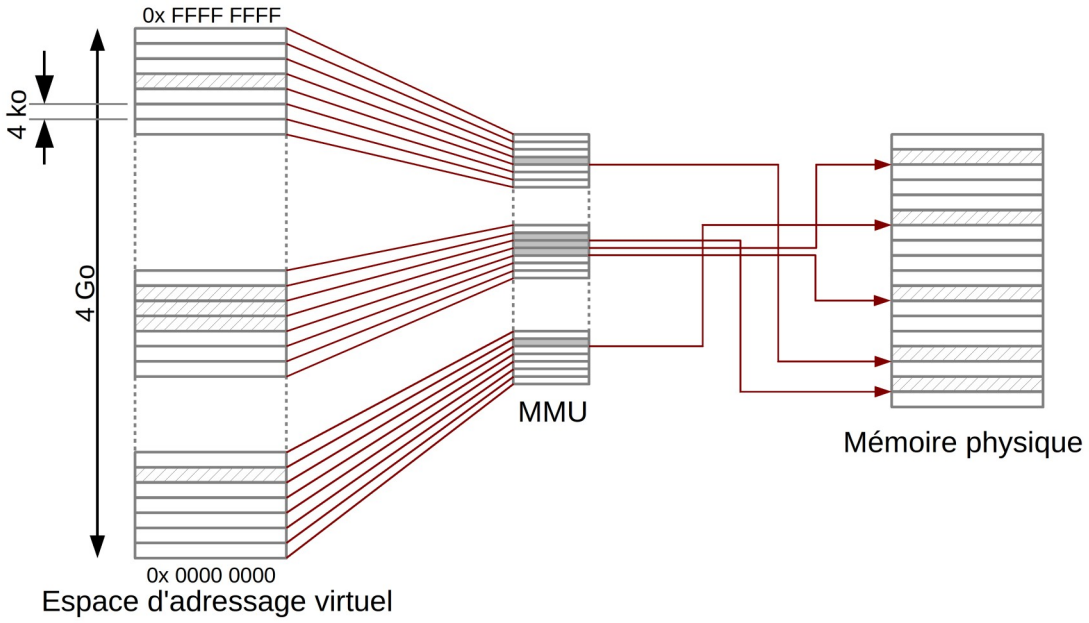
Les **processus** sont des espaces de mémoire disjoints, au sein desquels s'exécutent un ou plusieurs ***threads***.

L'ordonnanceur ne s'intéresse qu'aux tâches – les *threads*.

C'est le sous-système *Memory Managment* qui est concerné par la commutation entre processus.

La répartition des tâches en un ou plusieurs processus doit être décidée à la conception du produit.

La mémoire virtuelle



Un processus voit un espace de **mémoire virtuelle**, au sein du quel il peut accéder à n'importe quelle adresse de `0x000000` à `0xFFFFFFFF` (sur processeur 32 bits).

Cet espace est découpé en pages, et la **MMU** – *Memory Managment Unit* (un composant intégré dans le processeur) – associe une page de mémoire virtuelle avec une page de mémoire physique en effectuant la modification d'adresse lors de l'accès à la mémoire.

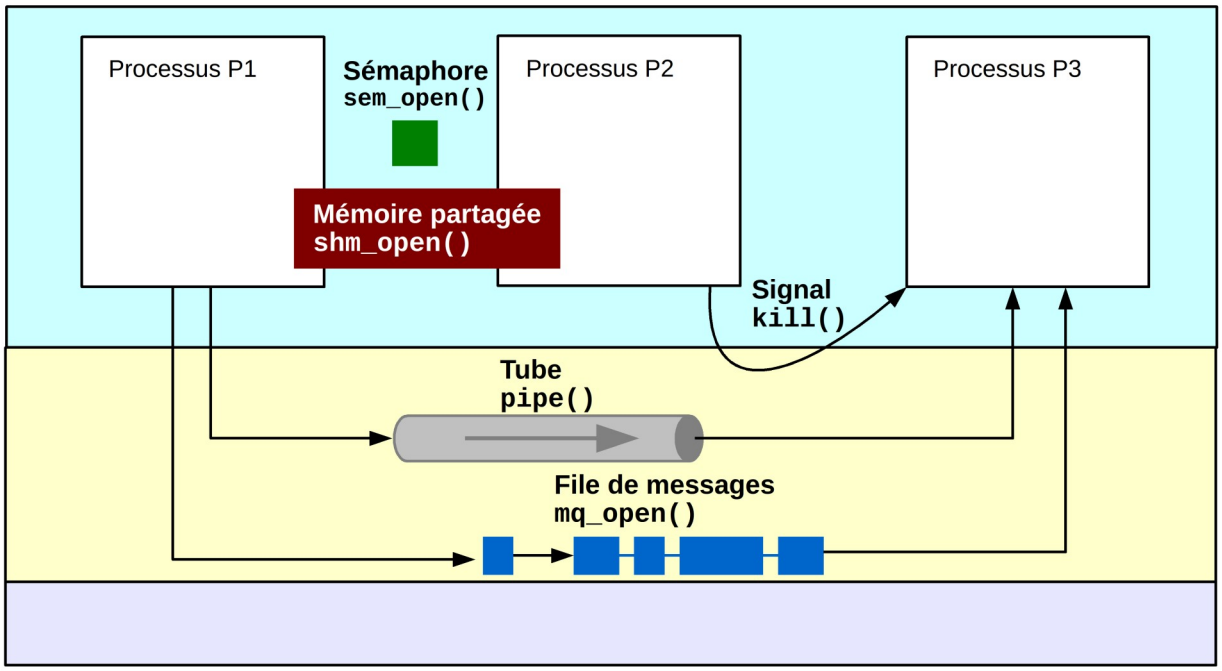
Certaines pages de mémoire virtuelle n'ont pas de correspondance en mémoire physique : une tentative d'accès déclenche une interruption « **faute de page** ».

Chaque processus dispose d'une configuration personnelle de la MMU. Cette dernière est programmée à chaque commutation entre deux processus.

Un processus ne voit que les pages de mémoire physique qui lui ont été attribué par le noyau ; les pages des autres processus ne sont projetées à aucun emplacement de sa mémoire virtuelle.

Communications entre processus

Mécanismes IPC (*Inter Processes Communication*)



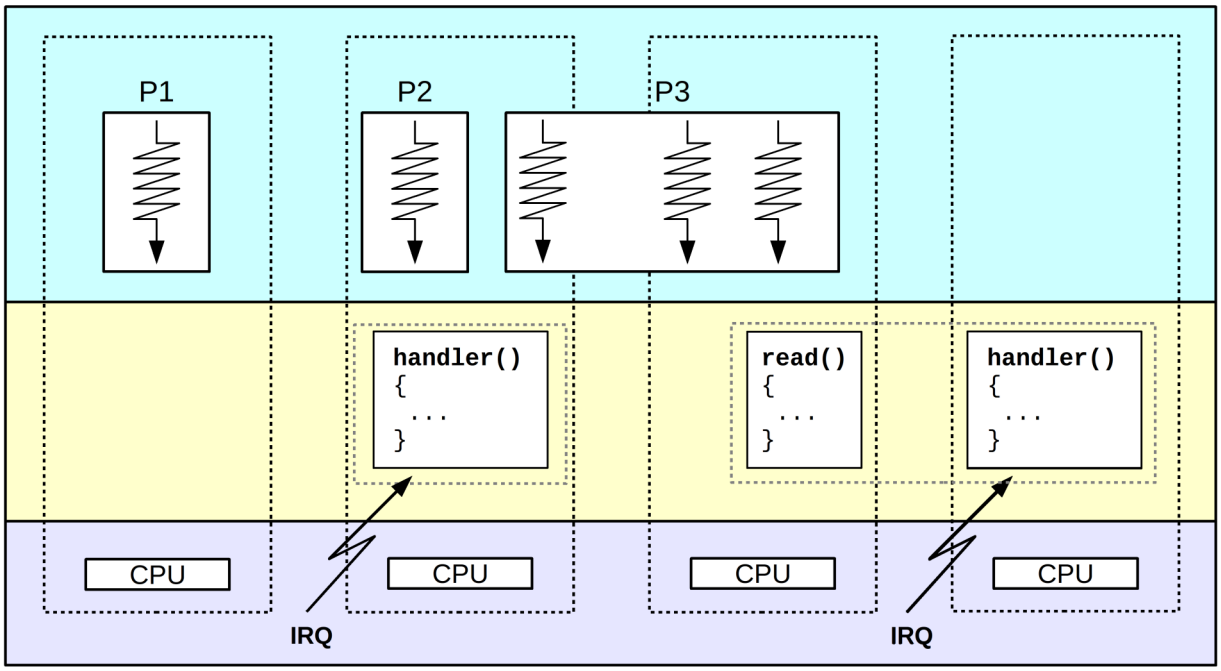
Les mécanismes de communications principaux entre les processus sont :

- la **mémoire partagée** : la même zone de mémoire physique est projetée dans la mémoire virtuelle des deux processus ;
- les **sémaphores** : qui servent surtout à synchroniser l'accès aux données de la mémoire partagée ;
- les **tubes** : un processus peut envoyer dans un tube des octets qu'un autre processus viendra lire ;
- les **files de messages** : les messages sont des blocs de données qui conservent leur cohésion lors du transfert entre deux processus, et qui peuvent être dotés de priorités ;
- les **signaux** : permettent à un processus de notifier un autre processus de l'occurrence d'un événement.

Pour tous ces mécanismes de communication, il est possible d'organiser des attentes passives (opérations bloquantes) de changement de conditions.

Programmation multicœur et temps réel

Répartition des tâches et traitements d'interruption



Il est aisément possible sous Linux :

- de choisir sur quel CPU (ou ensemble de CPU) une **tâche** sera exécutée grâce à la commande `taskset` ou aux fonctions `sched_setaffinity()` et `pthread_setaffinity_np()` ;
- de contraindre certaines tâches à utiliser uniquement certains CPU et certains **nœuds de mémoire** (sur les système NUMA *Non Uniform Memory Access*) avec le pseudo-système de fichiers `cpuset`.
- de laisser le noyau choisir le CPU le plus adapté (**migration** à volonté) ;
- de décider de traiter certaines **interruptions** sur un CPU (ou un ensemble) avec l'interface `/proc/irq/` ;
- d'interdire l'usage de certains CPU à tout le système afin de les laisser à la disposition d'un système externe avec l'option de démarrage `isolcpus`.

Temps réel avec Linux

Sous **Linux standard** (*Vanilla*), une tâche même très prioritaire peut être gelée le temps de traiter une interruption (parfois longue).

Avec un noyau Linux contenant le *patch* **`PREEMPT_RT`**, une tâche peut avoir une priorité supérieure à celle des traitements d'interruption.

Avec **Xenomai**, une tâche peut être plus prioritaire que tout le noyau Linux.

Version	**Linux standard**	**Linux `PREEMPT_RT`**	**Xenomai**
Latence des interruptions	~ 10 microsecondes	~ 10 microsecondes	~ 10 microsecondes
Latence d'interruption max.	< 1 ms.	< 500 microsecondes	< 50 microsecondes
Période mini timers	~500 µs	~100 µs	~10 µs
Retard maxi timers	< 2 ms.	< 500 µs	Qq dizaines de µs.

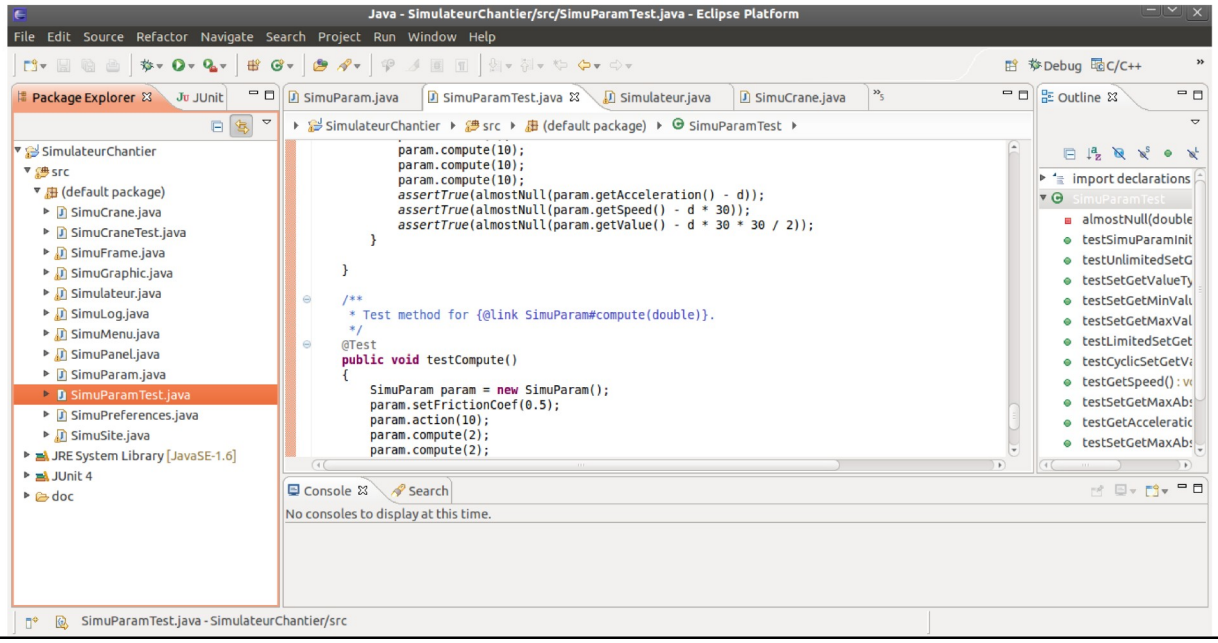
Le paramétrage de la priorité temps réel se fait avec :

- la commande shell `chrt`,
- les fonctions `sched_setscheduler()` et `pthread_setschedparam()`.

Environnement Linux et méthodes logicielles

Outils de développement libres

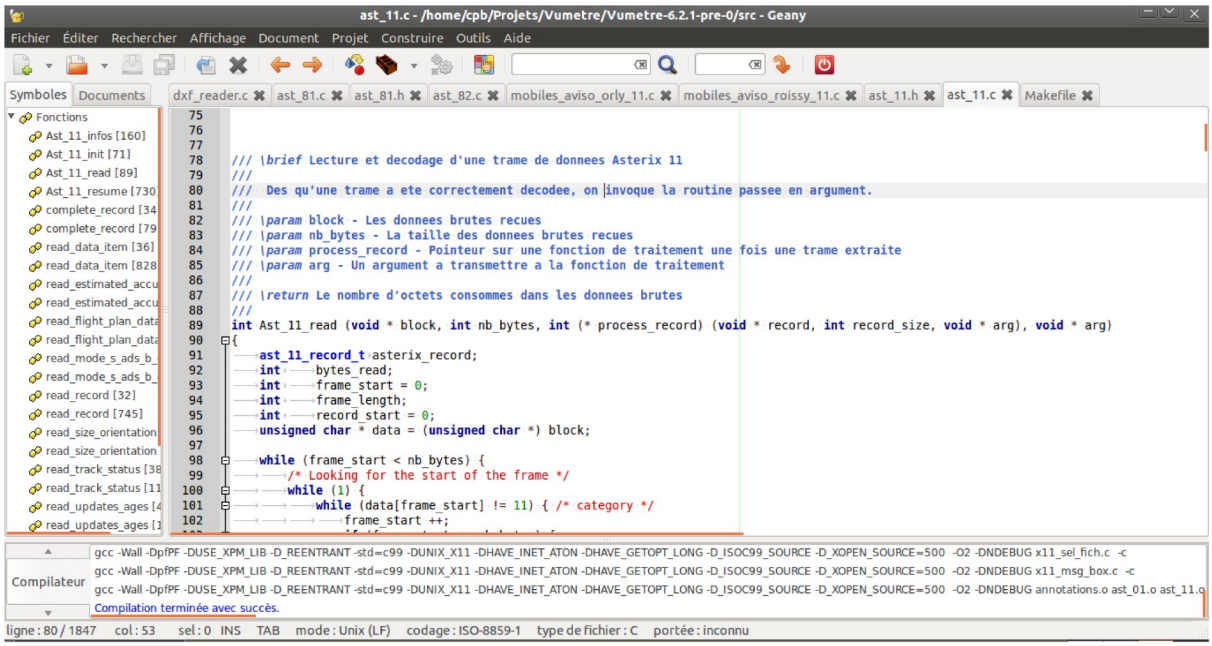
Éclipse



Éclipse est un environnement de développement intégré permettant de gérer des projet Java, C, C++, Python, etc.

Éclipse permet de réaliser du débogage local ou sur une cible distante.

Geany



Geany est un IDE plus simple et plus léger qu'Éclipse. Des extensions permettent de lui faire gérer des projets complets.

Langages et outils de développement

Compilateurs :

- C, C++, Objective C, Fortran, Ada, Java, etc.

Interpréteurs :

- Shell, Perl, Python, Java, Ruby, Tcl/Tk, etc.

Outils de mise au point :

- débogueur, archiveur, profiler, couverture de code, etc.

Suivi de version :

- Subversion, Git, Mercurial, etc.

Tous ces outils sont disponibles librement dans toutes les distributions Linux.

Licences GPL et LGPL

La plupart des logiciels fournis dans l'environnement Linux sont distribués suivant les termes de la licence **GPL** (*General Public License*) :

- le logiciel est libre suivant les critères de la *Free Software Fundation* (projet GNU)
- la clause du *copyleft* couvre la redistribution du logiciel.

Un logiciel est *libre* si son utilisateur dispose des quatre droits suivants :

- exécution et utilisation du logiciel ;
- analyse et adaptation du logiciel suivant ses propres volontés ;
- redistribution de copies non modifiées du logiciel ;
- redistribution de copies modifiées du logiciel.

Le *copyleft* précise que la redistribution du logiciel doit s'accompagner des mêmes droits que ceux concédés par l'auteur initial.

La licence LGPL est utilisée pour les bibliothèques : leur code interne est libre, mais elles peuvent être liées avec du code non-libre sans modifier la licence de celui-ci.

Conclusion et questions

Pour le programmeur la transition du micro-contrôleur vers le système Linux embarqué demande :

- la modification des boucles de **scrutation active** (*polling*) en **attentes bloquantes**,
- le découpage judicieux du code métier en **processus** et en **threads**,
- l'emploi des mécanismes de **communication entre tâches** adéquats,
- l'utilisation des ressources CPU (multicœur) et des bons niveaux de **priorité temps réel**,
- ...

Questions ?

N'hésitez pas à me contacter sur

christophe.blaess.fr

ou

www.logilin.fr

Pour en savoir plus

« *Développement système sous Linux* » – Christophe Blaess – 3ème édition – Eyrolles, 2011.

« *Solutions temps réel sous Linux* » – Christophe Blaess – Eyrolles, 2012.