Vers un système d'exploitation : Les processus et leur gestion

L'organisation d'un système d'exploitation

L'organisation d'un système d'exploitation est basée sur les idées suivantes.

- Un certain nombre de fonctions de base fort dépendantes des détails technologiques de la machine, en particulier les opérations d'entrée sortie, sont prises en charge par le système.

- Le système est un programme qui travaille dans son propre "contexte" que l'on essaie de maintenir aussi séparé que possible du "contexte" des programmes utilisateurs.

- La séparation des contextes permet de concevoir le système et les programmes utilisateurs de façon relativement indépendante. Ces programmes s'exécutent a tour de rôle, simulant une exécution simultanée.

Les interruptions et les trappes

- Pour qu'un système puisse fonctionner, il faut qu'à certains moments on puisse forcer l'exécution du système. Cela nécessite des mécanismes particuliers : les "interruptions" et les "trappes".

- Une interruption est un mécanisme qui permet lors de certains évènements de forcer un changement de programme exécuté.

- Une "trappe" est un mécanisme qui force un changement du programme exécuté lors de l'utilisation d'instructions spéciales qui ne font pas partie de l'ensemble d'instructions défini pour la machine.

- Nous allons adapter la machine ULg01 pour qu'elle puisse gérer des trappes et des interruptions.

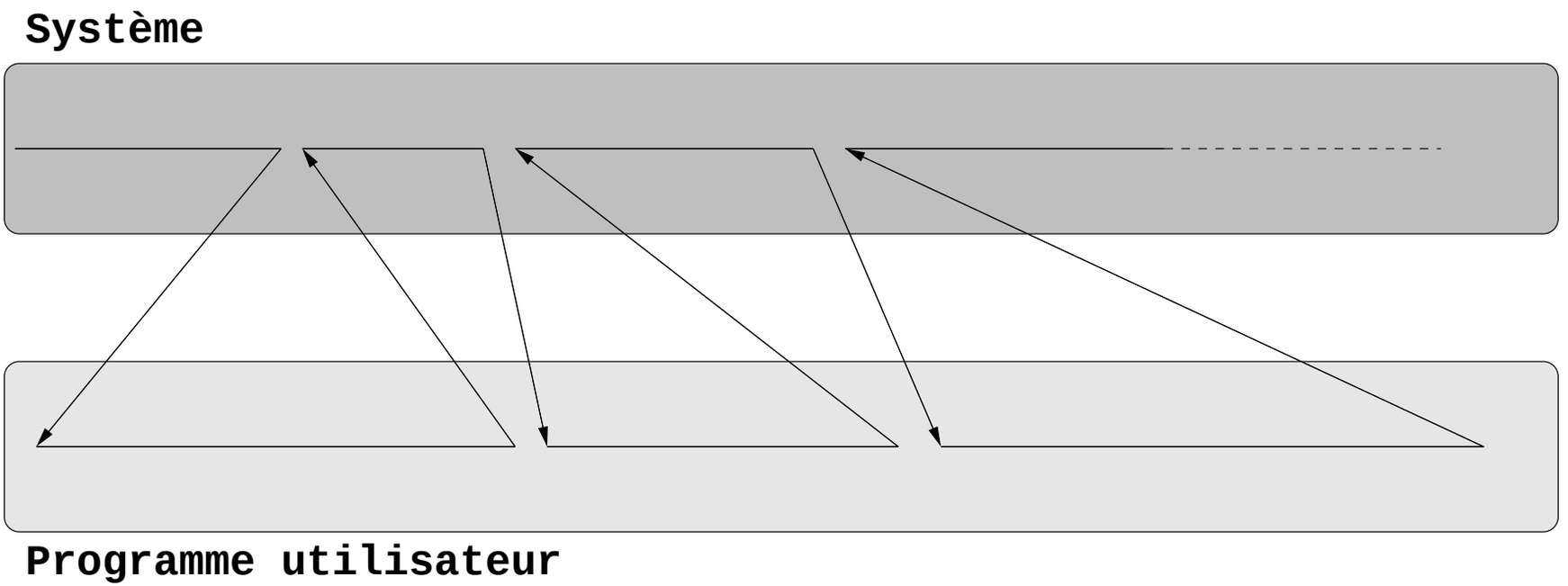
Fonctionnement du système : principes de base

Considérons d'abord la situation simple où il n'y a qu'un seul programme utilisateur en activité.

- Le but est de partager la machine entre le programme utilisateur et un "système" qui offre certaines fonctionnalités, notamment les entrées-sorties, au programme utilisateur.
- On exécute en alternance les deux programmes. Le passage du programme utilisateur au système se fait :
  - A la demande du programme utilisateur (trappe) ;
  - Lors d'une interruption ;
  - Lorsque le programme utilisateur exécute une opération illégale (faute).

  De façon générique, on appelle ces évènements des *exceptions*.
- Le passage du système au programme utilisateur se fait à l'initiative du système.

Une machine partagée



Actions lors d'une exception

Lors d'une exception, il faut

- Connaître l'adresse du programme système à exécuter,
- Préserver l'adresse de retour,
- Sauver l'état du programme utilisateur pour pouvoir restaurer cet état lorsque l'exécution du programme utilisateur reprendra.
- Lorsque le programme utilisateur reprend son exécution, il se retrouve dans la même situation qu'avant le saut au système. On dit que le programme utilisateur opère dans un *contexte* qui lui est propre.

Mode utilisateur - mode système

- Le but d'un système est d'isoler les utilisateurs des détails de la machine, mais aussi d'éviter que des programmes puissent bloquer la machine ou la mettre dans un état incohérent.

- Dans ce but, il est utile que certaines opérations ne soient possibles que pour le système.

- On prévoit donc un mode de fonctionnement "système" (parfois appelé mode "superviseur") et un mode de fonctionnement "utilisateur" dans lequel certaines opérations privilégiées ne sont pas possibles.

- Il est aussi habituel de désactiver les interruptions en mode "système".

Les choix de l'architecture β

- L'instruction d'opcode `0x00` sert d'appel au système. Des instructions spéciales d'opcode `0x0*` peuvent être définies, mais ne sont accessibles qu'en mode superviseur.

- L'adresse du programme système à exécuter lors d'une exception est fixée. Elle peut être différente suivant la cause de l'exception (interruption, trappe ou instruction illégale).

- Le PC du programme utilisateur est sauvé dans le registre 30, appelé `XP` - Exception Pointer, au moment d'un saut vers le système.

- Le bit 31 du compteur de programme sert d'indicateur de mode "superviseur".

- En mode "superviseur", les interruptions sont sans effet.

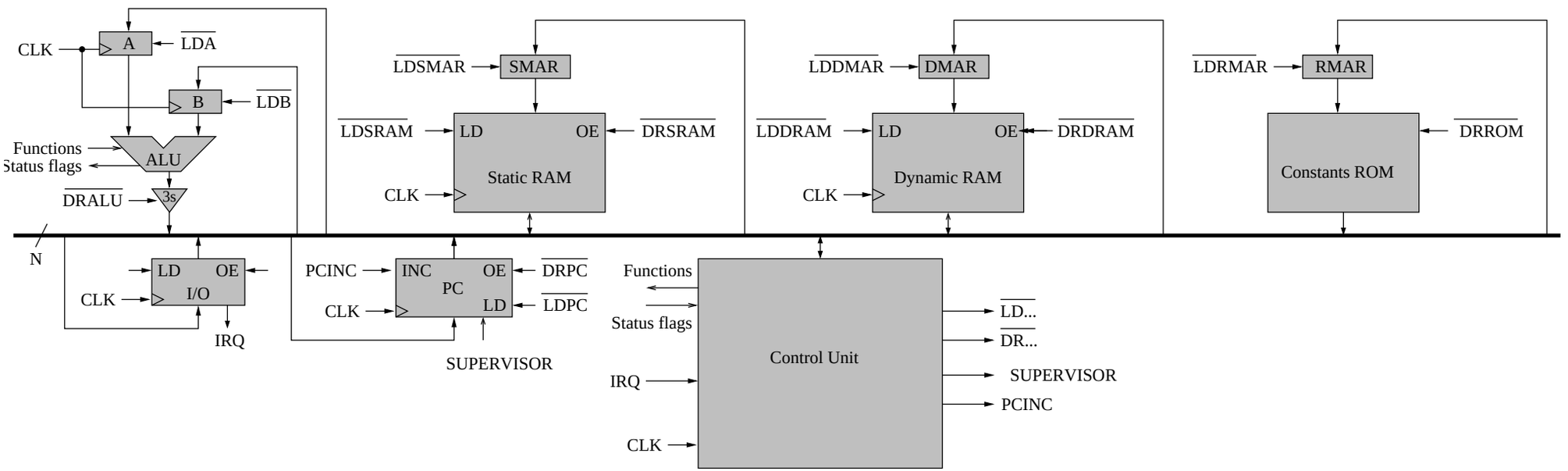
La machine ULg02

- Dans ULg02, deux nouveaux signaux sont utilisés en entrée de la ROM de microcode: `IRQ` et `PC31`. `PC31` est le bit de mode superviseur, `IRQ` est le signal d'interruption venant des dispositifs d'entrée-sortie.

- Un nouveau signal de contrôle `SUPERVISOR` sort de la ROM de microcode. Il sert à fixer la valeur du bit 31 du compteur de programme.

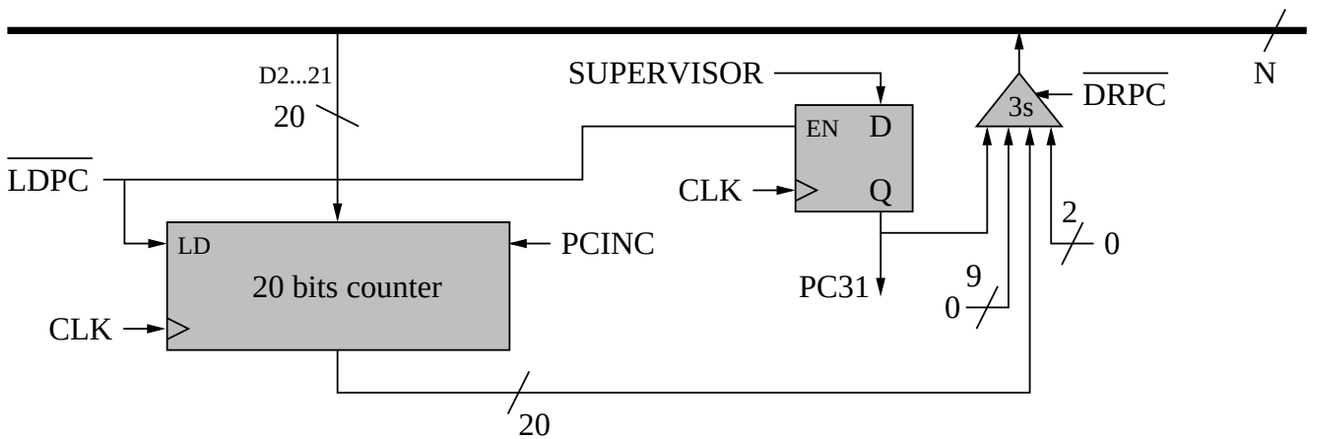
- Dans le microcode pour ULg02, il est utile de pouvoir disposer de constantes. Pour cela, on ajoute une ROM dans laquelle on place des constantes (par exemple l'adresse du gestionnaire d'interruptions) à partir des adresses les plus élevées. Pour contrôler cette ROM, il y a deux nouveaux signaux en sortie de la ROM de microcode : `LDRMAR` et `DRROM`.

ULg02 : une vue générale



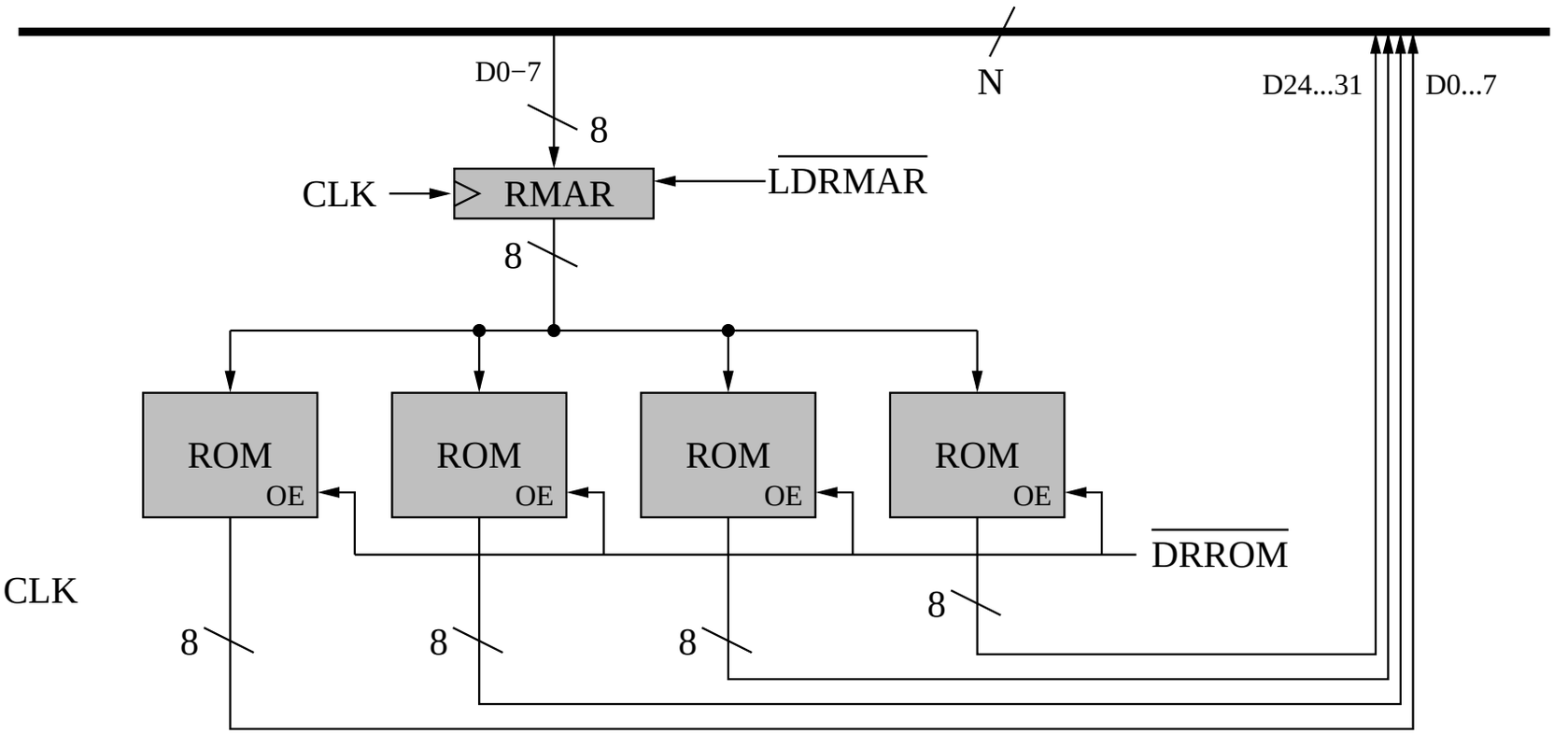
Remarquer le signal `SUPERVISOR` et l'ajout de la ROM de constantes

ULg02 : le compteur de programme (PC)



Le “bit de privilège” `PC31` est implémenté explicitement. Il ne peut être mis à 1 qui si le signal `SUPERVISOR` est à 1.

ULg02 : la ROM de constantes



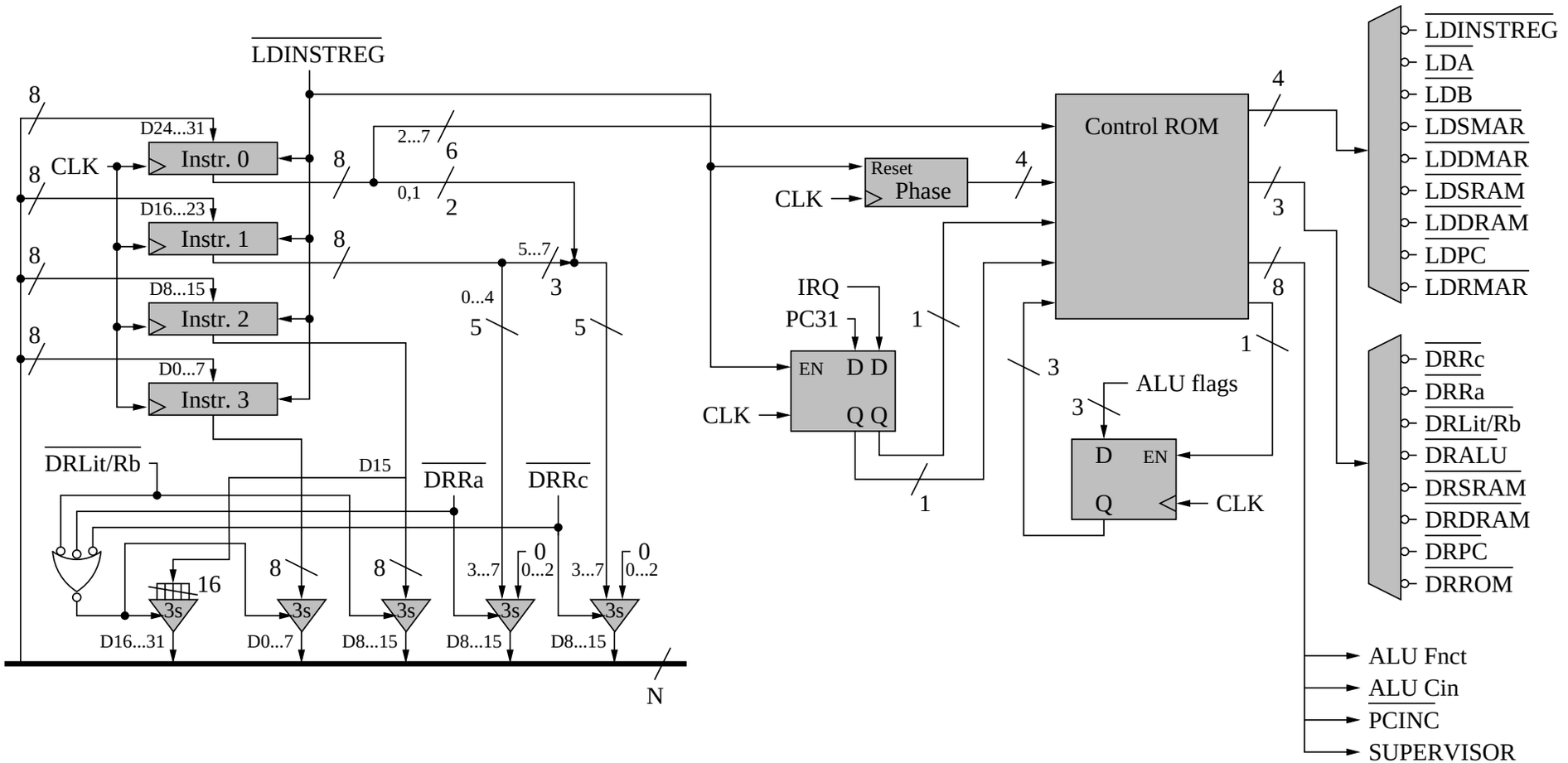
Les signaux contrôlant la ROM de constantes sont `LDRMAR` (chargement du registre d'adresse) et `DRROM` (lecture de la ROM).

ULg02 : le contenu de la ROM de constantes

0x00	⋮
	⋮
0xF9	10000000 00000000 01100000 00000000 *(Adresse du handler "Illegal Operation" — 0x6000)*
0xFA	10000000 00000000 01000000 00000000 *(Adresse du handler "IRQ" — 0x4000)*
0xFB	10000000 00000000 00100000 00000000 *(Adresse du handler "Supervisor Call" — 0x2000)*
0xFC	*(Adresse du handler "Cache Miss Code")*
0xFD	*(Adresse du handler "Cache Miss Data")*
0xFE	00000000 00000000 11110000 00000000 *(Adresse du registre XP)*
0xFF	*(Masque d'offset)*

Le *Masque d'offset* et les adresses des handlers *Cache Miss Code* et *Cache Miss Data* seront utilisés ultérieurement pour la gestion de la mémoire virtuelle.

ULg02 : l'unité de contrôle



La seule modification est l'ajout d'entrées (`PC31` et `IRQ`) et de sorties (`SUPERVISOR`, $\overline{\texttt{LDRMAR}}$ et $\overline{\texttt{DRROM}}$) à la ROM de microcode.

ULg02 : le microcode

- Pour toutes les instructions, lorsque IRQ et PC31 sont à 0, le microcode est simplement étendu en mettant SUPERVISOR à 0 ; $\overline{\text{LDRMAR}}$ et $\overline{\text{DRROM}}$ ne sont jamais utilisés.

- Lorsque PC31 est à 1, IRQ est sans effet (le microcode est le même que IRQ soit à 0 ou à 1). Le signal SUPERVISOR est toujours à 1 (on reste en mode superviseur), sauf lors d'un JMP. Dans ce cas, SUPERVISOR a, lors du chargement du PC, la valeur du bit de poids le plus fort de l'adresse de destination, ce qui permet de sortir du mode superviseur.

- Si PC31 est à 0 et que IRQ est à 1, quelle que soit l'instruction, on exécute le microcode spécial d'interruption qui sauve le PC dans XP, saute à l'adresse d'interruption contenue dans la ROM et met PC31 à 1.

- Il faut prévoir du microcode pour l'instruction d'appel au superviseur et pour les instructions illégales.

ULg02 : le microcode d'interruption

microcode IRQ : IRQ = 1 PC31 = 0 Opcode = ******

Phase	Flags	$\overline{\text{Latch flags}}$	ALU F, $\overline{C_{in}}$, Mode	LD SEL	DR SEL	PC+	SVR	
0000	*	1	110011	0001	011	0	0	A <- 0xFFFFFFFF
0001	*	1	111110	1000	011	0	0	RMAR <- A-1
0010	*	1	000000	0011	111	0	0	SMAR <- ROM
0011	*	1	000000	0110	110	0	0	SRAM <- PC
0100	*	1	111110	0001	011	0	0	A <- A-1
0101	*	1	111110	0001	011	0	0	A <- A-1
0110	*	1	111110	0001	011	0	0	A <- A-1
0111	*	1	111110	0001	011	0	0	A <- A-1
1000	*	1	111110	1000	011	0	0	RMAR <- A-1
1001	*	1	000000	0111	111	0	1	PC <- ROM
1010	*	1	000000	0100	110	1	0	DMAR <- PC; PC+
1011	*	1	000000	0000	101	0	0	INSTREG <- DRAM

Le PC sauvé est celui de l'instruction qui suit celle qui allait s'exécuter quand l'interruption est intervenue. L'instruction à l'adresse XP-4 n'a donc pas été exécutée.

ULg02 : le microcode de JMP

JMP(Ra,Rc) (mode utilisateur) : Opcode = 011011 IRQ = 0 PC31 = 0

Phase	Flags	$\overline{\text{Latch}}$ $\overline{\text{flags}}$	ALU F, $\overline{C_{in}}$, Mode	LD SEL	DR SEL	PC+	SVR	
0000	*	1	000000	0011	001	0	0	SMAR <- Ra
0001	*	1	000000	0001	100	0	0	A <- SRAM
0010	*	1	000000	0011	000	0	0	SMAR <- Rc
0011	*	1	000000	0101	110	0	0	SRAM <- PC
0100	*	1	111111	0111	011	0	0	PC <- A
0101	*	1	000000	0100	110	1	0	DMAR <- PC; PC+
0110	*	1	000000	0000	101	0	0	INSTREG <- DRAM

ULg02 : le microcode de JMP (suite)

JMP(Ra,Rc) (mode superviseur) : Opcode = 011011 IRQ = * PC31 = 1

Phase	Flags	$\overline{\text{Latch flags}}$	ALU F,$\overline{C_{in}}$,Mode	LD SEL	DR SEL	PC+	SVR	
0000	*	1	000000	0011	001	0	0	SMAR <- Ra
0001	*	1	000000	0001	100	0	0	A <- SRAM
0010	*	1	000000	0011	000	0	0	SMAR <- Rc
0011	*	1	000000	0101	110	0	0	SRAM <- PC
0100	*	0	110010	0010	011	0	0	B <- A+A; Latch
0101	$\overline{C}$=0	1	111111	0111	011	0	1	PC <- A
0101	$\overline{C}$=1	1	111111	0111	011	0	0	PC <- A
0110	*	1	000000	0100	110	1	0	DMAR <- PC; PC+
0111	*	1	000000	0000	101	0	0	INSTREG <- DRAM

L'instruction SVC

On ajoute à l'ensemble d'instructions de β une instruction d'appel au système.

```
.macro SVC()   LONG(0x0)
```

- Les arguments de cette instruction (nature de l'appel codée sur un mot et valeurs transmises éventuelles) sont placées sur la pile avant l'utilisation de l'instruction.

- Un appel au système peut renvoyer une valeur dans le registre 0 (c.f. un appel de procédure).

- L'exécution de `SVC()` sauve le `PC` dans `XP` et saute à une adresse contenue dans la ROM de constantes.

- En mode superviseur, l'instruction `SVC()` est possible, mais ne devrait pas être utilisée.

ULg02 : le microcode de SVC

SVC() : Opcode = 000000 IRQ = 0 PC31 = *

Phase	Flags	$\overline{\text{Latch flags}}$	ALU F, $\overline{C_{in}}$, Mode	LD SEL	DR SEL	PC+	SVR	
0000	*	1	110011	0001	011	0	0	A <- 0xFFFFFFFF
0001	*	1	111110	1000	011	0	0	RMAR <- A-1
0010	*	1	000000	0011	111	0	0	SMAR <- ROM
0011	*	1	000000	0110	110	0	0	SRAM <- PC
0100	*	1	111110	0001	011	0	0	A <- A-1
0101	*	1	111110	0001	011	0	0	A <- A-1
0110	*	1	111110	0001	011	0	0	A <- A-1
0111	*	1	111110	1000	011	0	0	RMAR <- A-1
1000	*	1	000000	0111	111	0	1	PC <- ROM
1001	*	1	000000	0100	110	1	0	DMAR <- PC; PC+
1010	*	1	000000	0000	101	0	0	INSTREG <- DRAM

Le PC sauvé est celui de l'instruction suivant SVC().

Les gestionnaires d'exceptions

- Les gestionnaires d'exceptions sont les points d'entrée dans le système à partir du programme utilisateur.

- La première chose qu'un gestionnaire d'exception doit faire est sauver l'état du programme utilisateur.

- L'état du programme utilisateur est le contenu des registres et de la mémoire qu'il utilise. Les registres seront sauvés. La mémoire utilisée doit être maintenue distincte de celle du système.

- Nous verrons par la suite une technique pour complètement séparer la mémoire du programme utilisateur de celle du système. Pour l'instant, supposons simplement que le système et le programme utilisateur ont chacun une zone mémoire qui leur est réservée.

Les gestionnaires d'exceptions (suite)

Un gestionnaire d'exception est écrit en deux parties :

- Un *stub* (*chicot*) écrit en assembleur et qui sauve l'état du programme utilisateur.

- Le gestionnaire lui-même qui peut être écrit sans précautions particulières et donc dans un langage de haut niveau (C par exemple).

Un “stub” de gestionnaire d’interruptions

L'état du programme utilisateur est sauvé dans la mémoire du système à une adresse fixée : `User`.

```
h_stub:  SUBC(XP, 4, XP)        | prévoir de  reprendre à
                                | l'instruction interrompue
         ST(r0, User, r31)      | sauver
         ST(r1, User+4, r31)
              . . .
         ST(r30, User+30*4, r31)
         CMOVE(KStack, SP)      | Charger le SP du système
         BR(Handler,LP)         | Appel du Handler
         LD(r31, User, r0)      | restaurer
         LD(r31, User+4, r1)
         LD(r31, User+30*4, r30)
         JMP(XP)                | retour à l'application
```

Pour un gestionnaire d'appel système on procède de même, mais on n'exécute pas `SUBC(XP, 4, XP)` car `XP` contient l'adresse de l'instruction suivante à exécuter.

Un système gérant des opérations d'entrée/sortie

- Supposons que ULg02 dispose d'une interface d'entrée-sortie simple connectée à un clavier.

- Du point de vue de la programmation, l'interface du clavier apparaît comme comme deux adresses en mémoire : `Data` et `Flag`. L'adresse `Data` contient le dernier caractère entré au clavier. L'adresse `Flag` contient 0 si aucun caractère n'est disponible et 1 si un caractère est disponible.

- Si `Flag` est à 1, le signal `IRQ` est actif.

- Il nous faut écrire un gestionnaire d'interruption et un gestionnaire de l'appel système "readkey".

Un gestionnaire d'interruption élémentaire

Le programme ci-dessous est le gestionnaire d'interruption du noyau du système.

```
struct Device { char Flag, Data; } Keyboard;
      /* La vue logique de l'interface d'entrée/sortie */
char Buffer[100];
      /* une zone de travail pour placer les caractères lus */
int inptr = 0;
     /* la place libre suivante */
int outptr = 0;
     /* Le caractère suivant à lire */

IntHandler()
{ Buffer[inptr] = Keyboard.Data;
  inptr = (inptr + 1) % 100;
  Keyboard.Flag = 0;
}
```

Grâce à l'interruption, chaque caractère est traité dès qu'il est disponible.

Le programme utilisateur est interrompu mais ne se rend compte de rien.

Un gestionnaire d'appel système

Le handler suivant est exécuté lorsque l'instruction `SVC` est invoquée avec le code "keyboard" placé sur la pile.

```
struct Mstate { int R0; ..., R30;) User;
       /* l'état sauvé du programme utilisateur */

KeyHandler()
{ while (ouptr == inptr) {};
  User.R0 = Buffer[outptr];
  outptr = outptr+1 % 100;
}
```

Un gestionnaire d'appel système amélioré

Le handler précédent ne fonctionne pas lorsqu'il est appelé avec `Buffer` vide car il ne peut pas être interrompu.

Si `Buffer` est vide, il faut revenir au programme utilisateur et refaire l'appel système.

```
KeyHandler()
{ if (ouptr == inptr) {
      User.R30 = User.R30 - 4;}  /* R30 correspond à  XP */
  else {
      User.R0 = Buffer[outptr];
      outptr = outptr+1 % 100;}
}
```

Ce programme crée une boucle qui est interrompue quand un caractère arrive. Pour faire mieux, il faut partager la machine entre plusieurs programmes utilisateurs.

Les processus

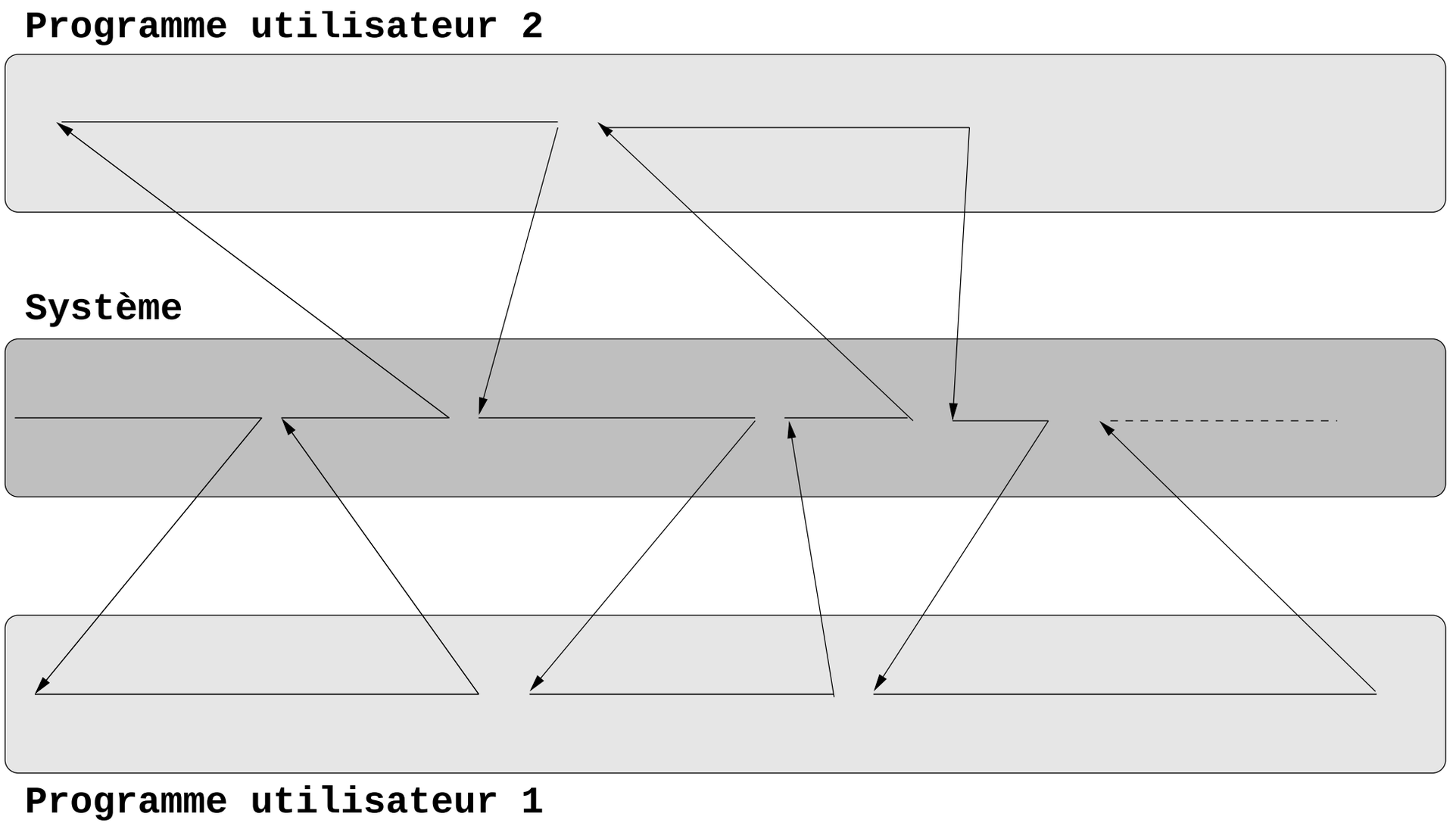
- Pendant qu'un programme utilisateur attend une opération d'entrée sortie, pourquoi ne pas permettre à un autre programme utilisateur d'utiliser la machine.

- On doit donc gérer un ensemble de programmes en exécution ou *processus*. Chaque processus a son espace mémoire réservé et quand il n'est pas actif, les valeurs qu'il a placées dans les registres sont sauvées.

- C'est le système qui gère les processus et qui, chaque fois qu'il a terminé ce qu'il à faire, exécute une fonction `scheduler()` qui choisit le processus qui sera réactivé.

- L'information concernant les processus est maintenue dans une *table des processus*.

Une machine partagée par plusieurs processus



Une gestion de processus élémentaire

```
struct Mstate { int R0; ..., R30;) User;
       /* l'état sauvé du processus  courant */

struct Mstate Proctbl[N]:
      /* La table des processus */

int Cur;  /* l'index du processus courant */

scheduler() {                    /* le gestionnaire de processus */
    Proctbl[Cur] = User:
    Cur = (Cur+1)%N;
    User = Proctbl[Cur];
}
```

Un gestionnaire d'appel système avec gestion des processus

Cette fois, si `Buffer` est vide, on rendra la main à un autre processus que celui qui a fait l'appel système.

```
KeyHandler()
{ if (ouptr == inptr) {
      User.R30 = User.R30 - 4;
      scheduler(); }
  else {
      User.R0 = Buffer[outptr];
      outptr = outptr+1 % 100;}
}
```

Les interruptions d'horloge

- Supposons que notre machine dispose d'une "horloge" sous la forme d'une adresse mémoire `clock` contenant une valeur incrémentée à chaque période de horloge hardware.

- Supposons aussi que cette horloge génère une interruption toutes les 10.000 périodes.

- Le gestionnaire des interruptions de l'horloge pourrait alors simplement être le suivant.

```
ClkintHandler()
{
   scheduler();
}
```

- Cela permet de garantir que `scheduler` est invoqué suffisamment souvent pour que chaque processus soit exécuté fréquemment.

Une gestion de processus plus évoluée

- Quand un processus attend une entrée sortie, il est inutile de le réactiver si rien ne s'est passé du côté du périphérique.

- Dans ce but, on note dans la table des processus un *état* du processus qui indique si ce processus est *actif* ou en *attente*. Lorsqu'un processus est en attente, un code indique la raison de sa mise en attente.

- La table des processus est alors définie comme suit.

```
struct PD {struct Mstate state ; int status} Proctbl[N]:
      /* La table des processus avec statut*/
```

- En prenant la convention qu'un statut 0 indique un processus actif et un statut 1 l'attente d'un caractère du clavier, nous pouvons réécrire notre gestionnaire d'entrée-sortie

Un gestionnaire d'interruptions avec gestion des processus et statut

```
struct PD {struct Mstate state ; int status} Proctbl[N]:
      /* La table des processus avec statut*/

IntHandler()
{ int i;
  Buffer[inptr] = Keyboard.Data;
  inptr = (inptr + 1) % 100;
  Keyboard.Flag = 0;
  for (i=0;i<=N;i++) {
      if (Proctbl.status[i]==1) Proctbl.status[i]=0;}
 }
```

On peut bien sûr éviter de parcourir la table des processus en utilisant des structures de données adéquates.

Un gestionnaire d'appel système avec gestion des processus et statut

Cette fois, si `Buffer` est vide, on rend inactif le processus qui attend un caractère du clavier.

```
KeyHandler()
{ if (ouptr == inptr) {
      User.R30 = User.R30 - 4;
      Proctable.status[Cur] = 1;
      scheduler(); }
  else {
      User.R0 = Buffer[outptr];
      outptr = outptr+1 % 100;}
}
```

Il faut aussi modifier `scheduler` pour qu'il n'active pas un processus en attente.