Session 4

Interruptions et signaux

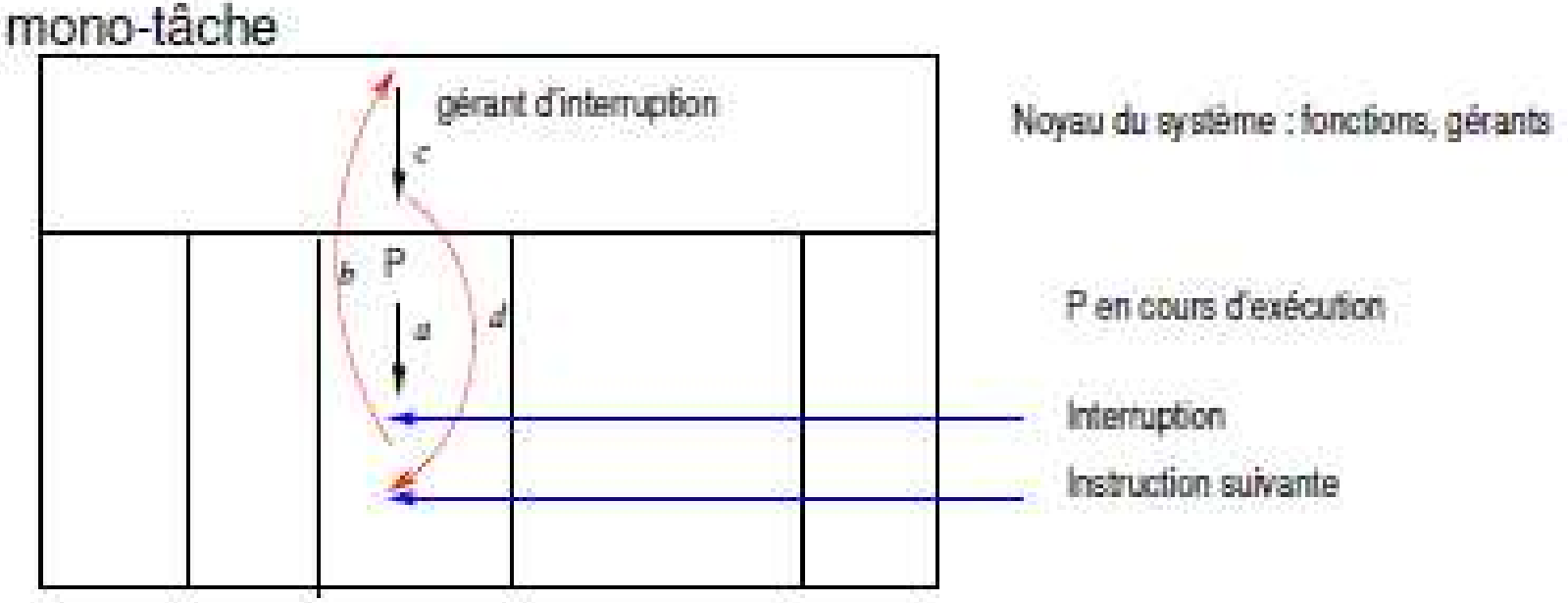
Introduction Interruptions

Interruption - système mono-tâche

- Mécanisme matériel, permettant la prise en compte d'un événement extérieur, **asynchrone**.

- Fonctionnement :
  1. l'instruction en cours est terminée
  2. sauvegarde du contexte minimal (CO, Mot d'État (PSW))
  3. une adresse prédéterminée est forcée dans CO; à partir de là tout se passe comme tout appel de fonction
  4. la fonction exécutée s'appelle gérant d'Interruption; on dit que le gérant correspondant à l'interruption est lancé
  5. fin du gérant et retour à la situation avant l'interruption : restauration du contexte précédent et suite de l'exécution

Schéma de fonctionnement d'une interruption



Interruption - Utilité

- Le mécanisme d'interruption est utilisé intensivement :
  - par les divers périphériques, afin d'alerter le système qu'une entrée-sortie vient d'être effectuée
  - par le système d'exploitation lui-même; c'est la base de l'allocation de l'UC aux processus. Lors de la gestion d'une interruption, le système en profite pour déterminer le processus qui obtiendra l'UC. Les concepteurs du système ont écrit les gérants des interruptions dans ce but.

- On en déduit que les systèmes multi-tâches vont détourner la dernière étape du déroulement de la gestion d'interruptions décrit précédemment.

Interruptions - Priorités

- Plusieurs niveaux de priorité des interruptions
- Un gérant d'interruption ne peut être interrompu que par une IT de plus haut niveau
- Une interruption de même niveau est masquée jusqu'au retour du gérant
- Ce mécanisme doit être assuré par le matériel : par exemple, IRET (Interrupt Return) permet de retourner d'un gérant donc de récupérer le contexte de l'interrompu !

Interruption - Exemple de priorités

- Un exemple dans l'ordre décroissant de priorité
  - Horloge
  - Contrôleur disque
  - Contrôleur de réseau
  - Contrôleur voies asynchrones
  - Autres contrôleurs...

- Exemples d'interruptions :
  - le contrôleur disque génère une interruption lorsqu'une entrée-sortie vient d'être effectuée
  - le clavier génère une interruption lorsque l'utilisateur a fini de saisir une donnée

Exceptions détectées et programmées

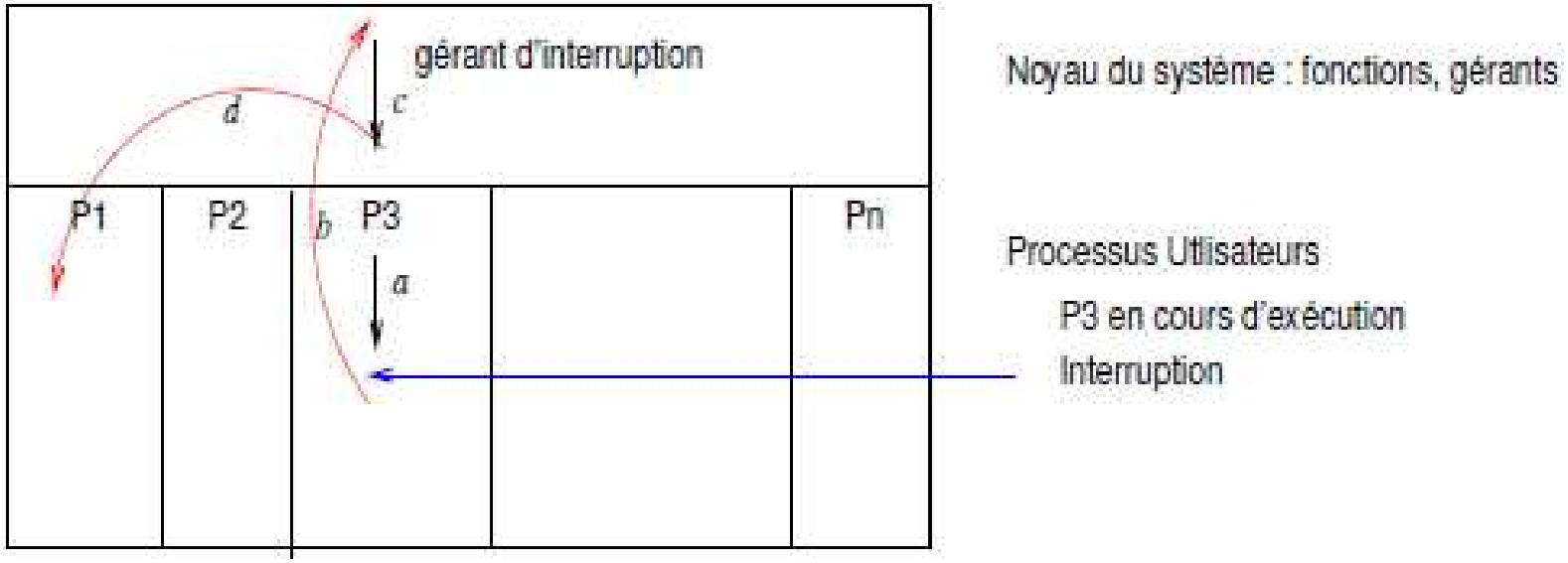
- Une exception génère un comportement semblable à celui des interruptions (gérant d'exception) mais elle est synchrone

- 2 types d'exception :
  - Exception détectée par le processeur : division par 0, overflow, défaut de page, ... appelé aussi déroutement, trappe ;
  - Exception programmée : elle permet d'implémenter un appel système grâce à une instruction privilégiée (interruption logicielle)

Horloge - Timer

- Le **temps partagé** permet aux processus de se partager l'UC. Ce mécanisme est réalisé grâce aux interruptions Horloge qui ont lieu périodiquement à **chaque quantum de temps**
- En quelque sorte, une minuterie qui est enclenchée au début de chaque processus et qui se déclenche toutes les 10ms
- Le gérant d'interruption de l'horloge va alors appeler **l'ordonnanceur** (scheduler) qui est chargé d'élire le prochain processus
- Les autres gérants d'interruption (E/S disque, ...) font de même

Ordonnancement des processus



Mode d'exécution

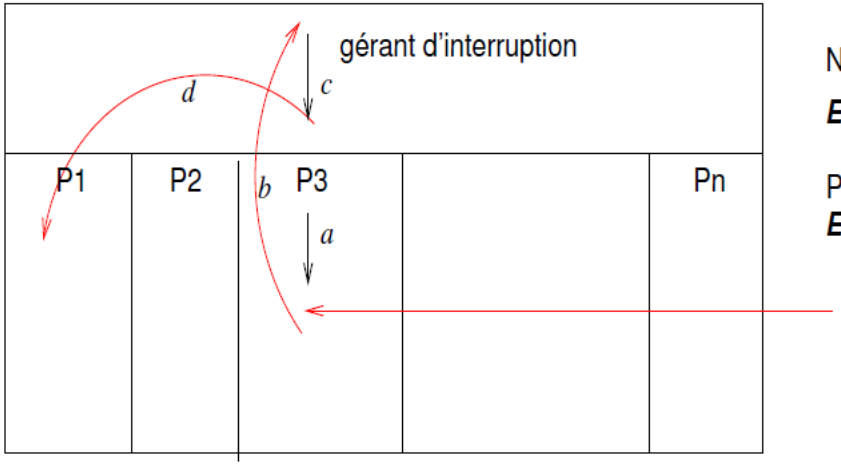
- 2 modes d'exécution alternatifs d'un même processus :
  - mode utilisateur : mode normal dans lequel les instructions machines du programme binaire sont exécutées
  - mode noyau : mode système ou privilégié où certaines instructions sensibles peuvent être exécutées

- Exemples d'instructions privilégiées : masquer les interruptions, manipuler l'horloge ou les tables systèmes, sortir de son espace de mémoire, . . .

- Raisons du passage en mode noyau :
  - Exception (détectée (déroutement), ou programmée (appel système))
  - Interruption

Changement de mode d'exécution

- Il effectue simultanément deux opérations :
  - bascule un bit dans le processeur (bit du mode d'exécution) ;
  - exécute le gérant d'interruption ou d'exception ;



Communication Entre Processus

- Jusque là chaque processus vivait de façon autonome, confiné et disposant seul de son espace mémoire. Mais cette solitude est trop restrictive; un besoin de communication avec les autres processus devient pressant.

- Les besoins de communications entre processus sont de deux ordres :
  - des données échangées entre processus, chacun les traitant à sa façon, un seul processus disposant de la donnée à un instant donné,
  - des données communes qu'ils pourraient partager, accéder et modifier en commun.

- Ces échanges et partages sont utiles tant pour les processus système qu'utilisateurs.

Exemples de Partages et Communications

- Une base de données gérant l'allocation de places à un spectacle est une donnée commune partagée entre plusieurs processus de réservation.

- Lorsqu'un processus parent prend connaissance de la fin d'un enfant, il y a expédition d'une donnée par l'enfant à destination du parent.

- unePatate | chaude | pourToi

est un échange de données entre trois processus ; *unePatate* envoie ses résultats à *chaude*, qui les utilise pour fournir des résultats à *pourToi*, qui les utilise, on se demande pourquoi, puisqu'on ne le dit pas ici

Types de Communications

- La forme des échanges ou partages peut être :
  - simple : l'échange est réduit à une occurrence ou un seul élément, expédié par un processus à destination d'un autre ; par exemple :
    - l'attente parent < - > enfant ;
    - un événement : une interruption ou un signal qui déclenchent une action (un gestionnaire).
  - complexe : échanges ou partages continus de données avec gestion de la protection et synchronisation.

Exemples

- Cas appelés simples :
  - wait() et waitpid() permettent à un parent d'attendre la fin d'un ou plusieurs descendants ;
  - kill est une commande envoyant une information (appelée signal) à un processus.

- Cas plus complexes :
  - unePatate|chaude|pourToi revient à lancer trois processus en reliant la sortie standard de *unePatate* avec l'entrée standard de *chaude* ainsi que la sortie standard de chaude avec l'entrée standard de *pourToi*.
  - On verra que le système prend en charge la synchronisation, par exemple, ne pas annoncer à *chaude* ou à *pourToi* qu'il n'y a plus de données si *unePatate* n'a pas fini son exécution.

Éléments de Solution

- Pour résoudre ces problèmes, il faut :
  - des moyens de communication, données ou structure de données que les processus peuvent échanger ou auxquelles ils peuvent accéder
  - des primitives d'accès et de protection
  - des mécanismes de synchronisation

- Le système peut assurer dans certains cas la prise en charge de la protection ou de la synchronisation, soulageant d'autant les processus utilisateurs (et les programmeurs). Mais lorsqu'il ne peut le faire, les processus utilisateurs en auront la charge.

Échanges Parent - Enfant

- Principe (sous Unix) : Tout processus qui se termine annonce à son générateur sa fin.
- Le générateur peut :
  - prendre connaissance de cette fin avec les primitives *wait()* ou *waitpid()* et analyser des informations relatives à cette fin (s'est-il terminé normalement ? ...) ;
  - ignorer toute fin de descendant.
- L'enfant est dans un état zombi (appelé aussi defunct dans certains systèmes) à partir de sa terminaison jusqu'à la prise en considération par le parent. Ce dernier peut être terminé.
- Noter que l'enfant ne sait pas que son parent se fiche de sa fin, c'est pourquoi il sera dans un état zombi tant que le parent n'est pas terminé.

Interruptions et signaux

Interruptions

- Le système d'interruptions (IT) joue un rôle prépondérant dans les systèmes d'exploitation; c'est grâce à lui que la multiprogrammation, ou traitement multi-tâche, est rendue possible.
- À l'origine il était destiné à améliorer le rendement de l'unité centrale (UC ou CPU) en facilitant considérablement la gestion des périphériques; puis d'autres fonctions lui ont été dévolues.
- L'asynchronisme de l'exécution des programmes par le CPU et des opérations d'Entrées/Sorties (E/S) au niveau des périphériques a été l'un des premiers problèmes que les concepteurs d'ordinateurs ont eu à résoudre.
- Dans les premiers SE, ce problème était entièrement pris en charge par le CPU qui devait scruter l'état du périphérique pour détecter la fin de l'E/S. cette technique de scrutation (ou polling) engendre une perte de temps CPU considérable
- Pour éviter ce situations, en découplant l'activité du CPU de celles des unités d'E/S, il s'est avéré nécessaire d'avoir recours à un mécanisme de bas niveau (hardware): le **mécanisme d'interruption**.

Interruptions

- D'une façon générale, le système d'interruptions peut être vu comme un **mécanisme de synchronisation**, en ce sens qu'il permet de résoudre le problème d'asynchronisme et dont le rôle consiste à indiquer au système d'exploitation l'occurrence d'un événement interne ou externe, attendu ou non.

- L'utilisation de ce concept a ensuite évolué pour prendre en charge non seulement l'événement que constitue la fin d'E/S mais aussi de nombreux autres événements que nous verrons plus loin.

Niveaux d'interruption et priorités

- Le système d'IT est réalisé en majeure partie par hardware et peut prendre en charge plusieurs types d'événements appelés **niveaux d'interruption**. Un périphérique signalera son événement, qui sera appelé **demande d'interruption**(1), sur le niveau qui lui est affecté. On peut associer à chaque niveau une priorité selon l'importance de l'événement qu'il représente; cette priorité permet de régler les conflits lorsque deux ou plusieurs niveaux d'interruption sont excités simultanément. On dit alors que le système d'interruption est hiérarchisé.

- (1) L'unité d'E/S demande au CPU d'interrompre l'exécution du programme en cours pour traiter son événement. Ceux qui sont plus familiers à la configuration de PC utilisent plutôt le terme IRQ (Interrupt Request)

Niveaux d'interruption et priorités

- La prise en compte d'une interruption provoquant la suspension de l'exécution du programme en cours et le lancement d'un programme approprié (appelé **routine d'interruption**) qui doit traiter l'événement signalé, constitue ce qu'on appelle une commutation de contexte au niveau du CPU. Le contexte d'un programme étant l'ensemble des informations représentant l'état d'avancement de son exécution à un instant donné;

- Il est constitué principalement de l'adresse stockée dans le compteur ordinal', de la valeur des indicateurs mémorisée dans le registre d'état, du contenu des registres de travail et des registres de base, etc. Le contexte du programme en cours d'exécution ou contexte courant se trouve dans le **mot d'état (PSW)** du CPU, un registre spécial qui a souvent une taille supérieure à celle d'un registre général.

- Les routines d'interruption font partie du noyau du système et sont donc **résidentes** en mémoire; cela évite d'avoir à les charger depuis le disque au moment où arrive une demande d'interruption, et donc réduit le délai pour débuter le traitement de l'interruption.

- Pour pouvoir reprendre l'exécution du programme courant qui est interrompu le temps de traiter l'interruption, il est impératif de sauvegarder son contexte avant d'entreprendre ce traitement.

Déroulement d'une interruption dans un système non hiérarchisé

- Dans cette section, nous décrivons les systèmes non hiérarchisés bien que n'ayant plus cours, pour montrer leurs lacunes et leur complexité par rapport à un système. En effet, dans les premiers processeurs, le système d'interruptions n'était pas hiérarchisé : plusieurs signaux d'interruption provenant de sources différentes arrivaient sur un niveau unique. Le déroulement du traitement de l'interruption se faisait en suivant les étapes suivantes :
  1. Sauvegarde du compteur ordinal dans un emplacement réservé en mémoire centrale (que l'on appellera *Sauve_CO*).
  2. Branchement à une adresse fixe, point d'entrée du programme de traitement de l'interruption.
  3. Désactivation du système d'interruptions (pour ne pas écraser le contenu de *Sauve_CO* par l'arrivée d'une nouvelle interruption).
  4. Sauvegarde du reste du contexte du CPU (dans une table *Tab_Contexte* en mémoire).
  5. Reconnaissance de l'interruption : quelles sont sa provenance et sa cause ?
  6. Traitement proprement dit de l'interruption (exécution d'une routine spécifique pour répondre à l'interruption).

Déroulement d'une interruption dans un système non hiérarchisé

7. Restauration du reste du contexte depuis *Tab_Contexte*.
8. Restauration du compteur ordinal depuis *Sauve_CO*.
9. Réactivation du système d'interruptions.

Les étapes 7, 8 et 9 sont souvent réalisées en une seule opération appelée *acquittement* de l'interruption. Les étapes 1, 2 et 3 sont typiquement câblées et simultanées, tandis que l'étape 6 est typiquement programmée.

- Pendant que le système d'interruptions est désactivé, les nouvelles demandes d'interruption ne sont pas perdues : elles sont mémorisées et mises en attente dans un registre appelé registre d'interruption. La reconnaissance de l'interruption peut être faite de façon :
  - implicite par hard : dans ce cas, à chaque interruption doit correspondre un point d'entrée différent;
  - ou explicite par soft : tester l'indicateur i (bit i du registre d'interruptions de taille égale au nombre de niveaux d'interruptions) qui correspond à l'interruption i pour savoir si celle-ci est demandée ou pas. Dans ce cas, l'ordre de test des indicateurs implique une priorité tacite entre les différents niveaux.

- L'inconvénient d'un système d'interruption non hiérarchisé est qu'il ne tient compte de l'urgence de la prise en compte d'une interruption. Pendant le traitement d'une interruption, si une demande d'interruption plus urgente survient, elle ne peut être prise en compte rapidement (pas de possibilité de suspendre une routine d'interruption).

Déroulement d'une interruption dans un système d'interruptions hiérarchisé

- Tous les processeurs de nos jours ont un système d'interruptions hiérarchisé : à chacun des niveaux est associée une priorité et le traitement d'une interruption peut être interrompu par une interruption plus prioritaire. Le déroulement de l'interruption de niveau **n** est le suivant :
  1. Sauvegarde du compteur ordinal dans un emplacement réservé spécifique *Sauve_CO*$_n$
  2. Branchement à une adresse fixe spécifique *Adr_interruption*.
  3. Sauvegarde du reste du contexte dans une table Tab$_k$, (k étant le niveau d'interruption du programme interrompu) et chargement du contexte de l'interruption n depuis la table Tab$_n$.
  4. Traitement proprement dit de l'interruption **n**.
  5. Restauration du contexte interrompu depuis Tab$_k$.
  6. Restauration du compteur ordinal depuis *Sauve_CO*$_n$,..

- Les étapes 1 et 2 tout comme les étapes 5 et 6 sont typiquement câblées et simultanées

Masquage des interruptions

- Les priorités associées aux interruptions sont figées par le hard. Cependant, on peut avoir besoin dans certaines applications de modifier ces priorités.
- Autrement dit, on ne veut pas être interrompu par une (ou plusieurs) interruption(s) donnée(s) pendant l'exécution d'un programme utilisateur ou système. Il existe pour cela un registre appelé registre masque, de taille égale au nombre de niveaux d'interruption, dans lequel on peut positionner un bit pour empêcher la prise en compte d'une interruption donnée.
- À la fin de son exécution (ou avant), le programme qui a masqué un (ou plusieurs) niveaux d'interruption doit remettre l'état du registre masque tel qu'il était avant le masquage.
- Dans la figure suivante, des demandes d'interruption sont arrivées sur les niveaux IT_6, IT_5 et IT_3. Les niveaux IT_5 et IT_4 sont masqués.

Conditions de prise en compte d'une interruption

- Une demande d'interruption n'est pas automatiquement traitée dès qu'elle survient (Sinon ce serait la dictature des interruptions).

- Le système d'interruptions teste d'abord (par hard) si toutes les conditions nécessaires à son traitement sont réunies. Ces conditions sont les suivantes :
  - l'interruption incidente n'est pas masquée ;
  - le CPU est à un point interruptible;
  - il n'y a pas de niveau d'interruption plus prioritaire en attente ou en cours de traitement.

- Si toutes ces conditions sont satisfaites, il provoque la commutation de contexte: le contexte du programme en cours d'exécution est sauvegardé en mémoire et le contexte de la routine de traitement de l'interruption est chargé dans le CPU,

- **Définition du point interruptible.** Le CPU possède un état observable (donc mémorisable) seulement à des instants précis où les valeurs dans les registres spéciaux (registre d'état, compteur ordinal, registre d'adresse mémoire, registre de données mémoire...) et généraux sont valides (stables).

Conditions de prise en compte d'une interruption

- Pendant l'exécution d'une instruction (qui se fait en plusieurs cycles), le contenu de ces registres n'est pas stable; il ne le sera qu'à la fin de l'exécution de cette instruction. On définit donc le point interruptible comme étant la fin de l'exécution d'une instruction et avant le début de l'exécution de la suivante.

- L'exécution d'une instruction se termine par le positionnement d'un indicateur dans la micromachine qui sera testé par le système d'interruptions pour détecter le point interruptible.

- Dans certaines instructions itératives ayant un temps d'exécution relativement important (instructions de mouvement de chaînes d'octets, par exemple), le point interruptible est généré à la fin de chaque itération (mouvement d'un octet).

Types d'Interruptions

- On regroupe sous le nom d'interruptions un certain nombre d'événements ayant des causes très différentes. Selon ces causes, on peut classer les interruptions en 4 catégories :
  1. interruptions dues aux erreurs matérielles ;
  2. interruptions provenant de l'exécution du programme ;
  3. interruptions causées par la terminaison des E/S ;
  4. interruptions provenant des organes externes (autres que les périphériques d'E/S).

- Les interruptions des deux premières catégories, ayant des causes étroitement liées au fonctionnement de la machine (CPU et mémoire), portent un autre nom : elles sont appelées **déroutements** (ou traps ou encore exceptions).

- Voyons maintenant quelles interruptions classer dans chaque catégorie:
  - **interruptions dues aux erreurs matérielles** : ces erreurs sont des défaillances techniques telles que le défaut d'alimentation (chute de tension du courant électrique) ou un mauvais fonctionnement de la mémoire (erreur de parité mémoire détectée lors du contrôle de parité dans une opération de lecture). Ces interruptions ne sont pas masquables ;

Types d'Interruptions

- **interruptions dues à une erreur du programme** : pendant l'exécution d'un programme utilisateur, certaines erreurs d'exécution peuvent être détectées par le hard, par exemple :
  - erreur arithmétique (division par 0, débordement arithmétique...),
  - instruction inexistante (code opération non reconnu : erreur généralement due à un mauvais calcul d'adresse dans une instruction de branchement),
  - tentative de violation de protection mémoire
  - instruction privilégiée en mode asservi! (appelé aussi mode utilisateur ou esclave), etc.;
    Remarque. La routine de traitement de ces déroutements signale l'erreur par un message, qui peut être accompagné de la valeur des registres au moment de l'erreur, et arrête l'exécution du programme (abort).

Types d'Interruptions

- **interruptions volontaires du programme** : il existe un déroutement particulier qui est provoqué volontairement par le programmeur pour demander un service du superviseur, le plus souvent une procédure du système qui réalise une E/S.
  Pour cela, le programme contient à l'endroit approprié une instruction d'appel au superviseur (SVC : SuperVisor Call) qui, lorsqu'elle est exécutée, provoque la commutation de contexte et l'exécution de la procédure système en question.
  Quand on programme dans un langage de programmation évolué et que l'on fait appel à certaines fonctions de la librairie ou du système, l'utilisation de SVC est transparente car celle-ci est incluse dans la fonction appelée;

- **interruptions causées par la terminaison des E/S** : ces interruptions sont transmises par les canaux d'E/S ou, à défaut, par les contrôleurs de périphérique pour signaler la fin d'une E/S. Ce sont les interruptions les plus fréquentes. Dans les systèmes multiprogrammés travaillant en batch, ces interruptions sont utilisées, en plus de leur fonction principale, pour provoquer la commutation de tâches utilisateur (chaque tâche ayant été affectée d'une priorité à sa création dans le système);

Types d'Interruptions

- **interruptions provenant des organes externes** (autres que les périphériques d'E/S): ces interruptions permettent de réaliser la communication extérieur-système.

- Dans cette catégorie, on distingue 3 interruptions principales auxquelles peuvent s'ajouter d'autres interruptions suivant la configuration matérielle de la machine
  - l'interruption Pupitre indiquant au système qu'il doit se mettre en état de réception de message de la console opérateur,
  - l'interruption Horloge générée à intervalles réguliers permettant de mesurer le temps réel,
  - les interruptions de comptage indiquant qu'une série de N événements identiques s'est produite (carte d'acquisition de données dans les systèmes industriels, par exemple).

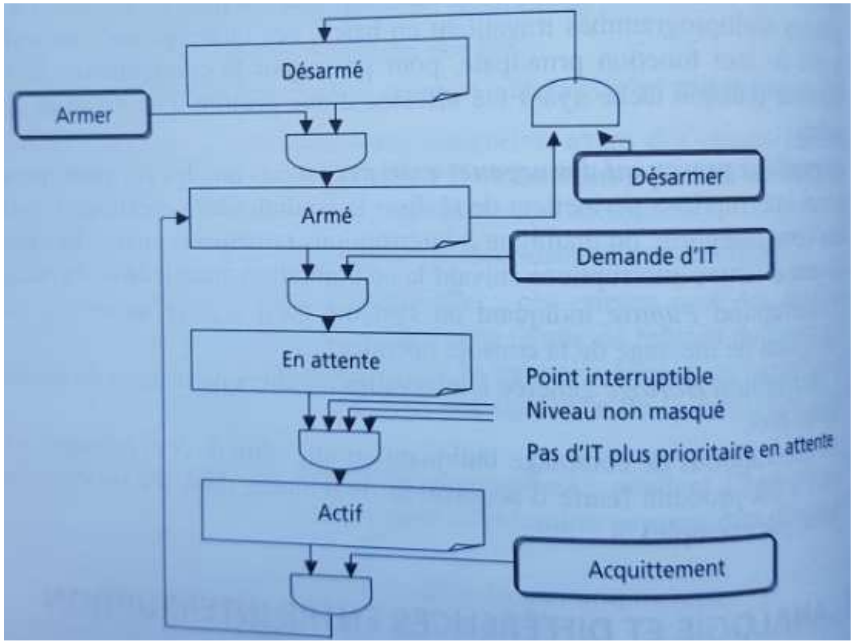
Analogie et différences entre interruption et déroutement

- La seule analogie entre interruption et déroutement est qu'il y a commutation de contexte dans les deux cas (le contexte du programme interrompu est sauvegardé, le contexte de la routine de traitement de l'interruption ou du déroutement est chargé dans le CPU).

- Les différences sont les suivantes :
  - un déroutement n'est pas masquable et ne peut être mis en attente :
  - un déroutement est toujours le résultat d'un événement dépendant du programme
  - l'instruction au cours de laquelle s'est produit le déroutement n'est pas toujours achevée (il n'y a pas d'incrémentation du compteur ordinal de sorte qu'après la traitement du déroutement, l'instruction est à nouveau ré-exécutée).

États d'un niveau d'interruption

- Une demande d'interruption passe par plusieurs états avant d'être réellement traitée. Le schéma suivant donne les transitions entre ces états (Désarmé, Armé, En attente, Actif) produites par les actions ou événements Désarmer, Armer, Demande d'IT, Acquittement.



Exemples d'utilisation des interruptions

- On s'intéressera ici surtout à l'interruption *Horloge*, les autres interruptions ayant un rôle bien figé. Sur toutes les machines, il existe une horloge (quartz + circuit intégré diviseur de fréquence) qui délivre un train d'impulsions avec une fréquence donnée.

- Ces impulsions arrivent sur un niveau d'interruption. Si T est la période de l'horloge (intervalle de temps séparant 2 impulsions d'horloge consécutives), alors on peut mesurer par soft tous les intervalles de temps d'une durée égale à k.T avec k entier positif.
- On utilise pour cela un registre spécial ou un mot mémoire dédié (que nous appellerons compteur), initialisé à la valeur k si l'on veut mesurer un intervalle de durée k.T, et décrémenté à chaque impulsion d'horloge. Lorsque le contenu du compteur devient nul, une interruption, appelée interruption de comptage, est générée par le hard.

- Dans la suite, il y a quelques exemples d'utilisation de l'interruption Horloge.

- Certains de ces exemples sont illustrés plus loin dans ce cours par des programmes en C tournant sous Unix.

Exemples d'utilisation des interruptions

- Prélèvement de mesures en contrôle de processus
  - Ce prélèvement est effectué régulièrement. S'il doit être fait toutes les 100 millisecondes et que l'horloge a une période de 5 microsecondes, il faudra initialiser le compteur à k = 20 000. Ainsi, le système d'interruptions reçoit une interruption de comptage toutes les 100 millisecondes. Le traitement de l'interruption de comptage doit se terminer par la réinitialisation du compteur à la valeur k = 20 000.

- Délai de garde (Time Out)
  - Il s'agit de contrôler le temps d'exécution d'un programme (ou d'une procédure) qui doit terminer son exécution dans un délai D. Si au bout du temps D, le programme ne s'est pas terminé, une interruption « Time Out » est générée et provoque l'exécution d'une routine d'interruption dont le rôle est de traiter l'erreur. La dernière instruction exécutée par le programme, lorsqu'il se termine normalement, désarme le niveau de l'interruption de comptage.

Exemples d'utilisation de déroutements

- Mesure de la taille d'une mémoire
  - La capacité mémoire maximale d'une machine donnée est atteinte en ayant K blocs physiques de 2^m octets. Sur certaines configurations de cette machine, le nombre de blocs physiques disponibles peut être inférieur à K. Dans les machines actuelles. PC ou serveurs, la capacité mémoire peut être étendue par une simple adjonction de barrettes mémoire dans des slots prévus à cet effet. Une des fonctions du programme de démarrage (BIOS) est de mesurer et d'afficher la taille de la mémoire disponible.
  - Le moyen simple d'y arriver consiste à utiliser les déroutements : on détermine la valeur de K par des tentatives d'accès successives au premier octet de chaque bloc et, dès que cette tentative provoque un déroutement (pour adresse inexistante), cela signifie qu'on a visité tous les blocs disponibles ; on a donc pu compter leur nombre.
  - La nouvelle routine de déroutement à mettre en place comporte l'incrémentation du nombre de blocs et la restauration du contexte interrompu. Lorsque K est déterminé, il faut réinstaller la routine standard du déroutement.

- Personnalisation du traitement des déroutements
  - Un programmeur système peut se trouver devant la nécessité de remplacer le traitement standard prévu par le système d'exploitation pour certains déroutements par des routines propres à lui. Il peut faire cela en remplaçant la première instruction des routines standard (dont il doit connaître l'adresse) par une instruction de branchement vers ses routines propres qui doivent se terminer par la restauration de contexte.
  - Il est possible qu'un déroutement (dont on a personnalisé le traitement) se produise à l'intérieur de sa routine de remplacement. Pour éviter d'entrer dans une boucle sans fin, il faut appliquer à ce déroutement le traitement standard.

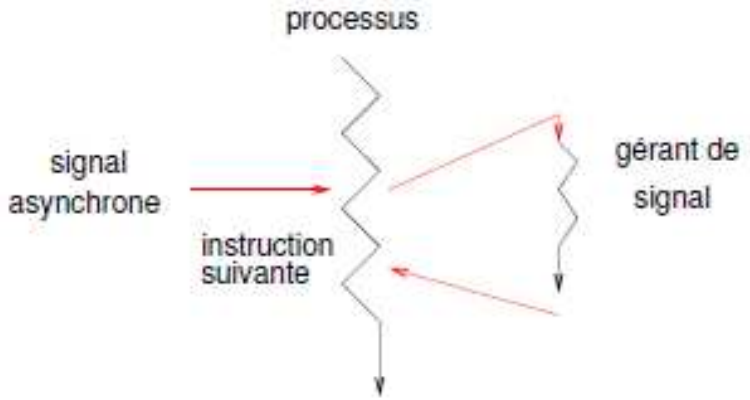
Signaux

Introduction

- Signal
  - Interruption: événement extérieur au processus
    - Frappe au clavier
    - Signal déclenché par programme: primitive kill, …
  - Déroutement: événement intérieur au processus généré par le hard
    - FPE (floating point error)
    - Violation mémoire

Introduction

- Caractéristiques des signaux :
  - Une forme d'interruption logicielle ; appelée ainsi à cause du comportement asynchrone : le processus ne sait ni s'il recevra ni quand il recevra un signal, d'où la similitude de fonctionnement avec une interruption matérielle



  - C'est un moyen pour expédier à un processus une information urgente
  - Toutes les informations urgentes sont connues et cataloguées, dans une liste complète et exhaustive du système

Signaux sous Linux

- Dans le système Unix, un signal désigne une interruption ou un déroutement. L'événement associé à un signal peut être:
  - soit un événement externe au processus (frappe au clavier, signal émis par un autre processus à l'aide de la primitive kill, signal d'horloge...);
  - soit un événement interne au processus, correspondant à une erreur (erreur virgule flottante, protection mémoire...).
- Il existe NSIG (<signal.h>) signaux différents, identifiés par un numéro de 1 à NSIG
- Comme pour les interruptions, on peut mettre en attente certains signaux en les masquant.
- Un processus qui reçoit un signal est terminé (ou aborté), à moins que ce signal soit intercepté ou ignoré (masqué), avec éventuellement génération d'un fichier de nom *core* qui contient son image mémoire au moment de l'abort.

Signaux sous Linux

- Un signal *pendant* est un signal en attente d'être pris en compte. Le bit associé dans le registre des signaux *pendants* est à 1. Si un autre signal du même type arrive alors qu'il en existe un *pendant*, il est perdu. Si le signal n'est pas masqué, sa prise en compte a lieu lorsque le processus passe à l'état actif *utilisateur*. Le bit *pendant* associé passe alors à 0.

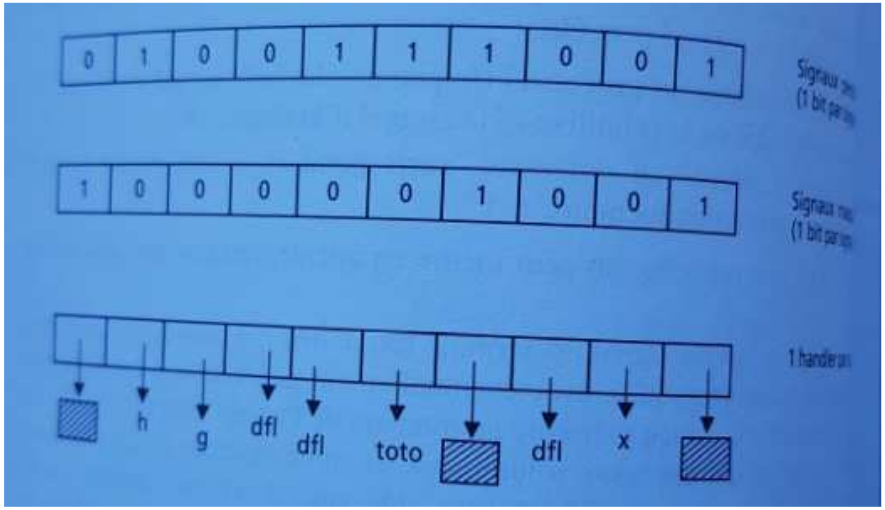
- Notons qu'un processus utilisateur en cours d'exécution est a l'état actif. S'il est en train d'exécuter une fonction du système, on dit qu'il est actif système ou actif noyau ; s'il est en train d'exécuter ses propres instructions, il est actif utilisateur. La prise en compte a lieu lorsque le processus passe de l'état actif noyau à l'état actif utilisateur : un processus en mode noyau ne peut être interrompu par un signal.

- La prise en compte d'un signal entraîne l'exécution d'une fonction spécifique appelée *handler*, Celle-ci peut être la routine prédéfinie dans le système (traitement standard ou par défaut) ou une routine mise en place par l'utilisateur pour personnaliser le traitement de ce signal.

Signaux sous Linux

- Dans le contexte du processus, appelé aussi bloc de contrôle (ou PCB pour Process Control Block), il existe une structure pour assurer la gestion des signaux qui regroupe la mémorisation des signaux en attente, le masquage et les pointeurs vers les *handlers*.

- On peut avoir par exemple la figure suivante dans laquelle *h*, *g*, *toto* et *X* sont des *handlers* installés par l'utilisateur, *dfl* est le *handler* par défaut et [carré grisé] le pointeur nul mis pour les signaux masqués.



Exemples

- Une erreur de segmentation provoque l'expédition d'un signal appelé SIGSEGV au processus fautif (actif), l'expéditeur étant une fonction système ;
- Ctrl C (contrôle C) génère l'expédition d'un signal, dont le traitement par défaut est l'arrêt du processus
- kill -9 32600 expédie le signal numéro 9 au processus numéro 32600. Le gérant du signal 9 consiste à détruire (terminer) le processus en question
- Il est préférable d'appeler les signaux par leurs noms plutôt que par leurs numéros. En effet, certains signaux portent des numéros différents selon le système d'exploitation.
  - Exemple : kill -HUP 32600 ou kill -SIGHUP 32600 sont préférables à kill -1 32600

Actions Possibles sur un Signal

- Un processus peut :
  - accepter l'action par défaut prévue (souvent arrêt), ou
  - gérer le signal (pas tous les signaux), ou
  - l'ignorer (pas tous).

- Dans tous les cas, une action par défaut est prévue dans le système. Cette action peut être d'ignorer le signal

- Seuls les processus issus du même propriétaire peuvent échanger des signaux

Gestion des Signaux

- Pour Programmer la gestion d'un signal il faut :
  - écrire la gérante(i.e. la fonction gérante),
  - faire l'association entre l'identité du signal et la fonction. Cette association permet d'indiquer au système la fonction à exécuter si le signal se produit.

- Forme générale des programmes :

```
void gerante (int numeroSignal){
//contenu de la gérante
}
int main(){ ...
associer (nomSignal, gerante) ;
//à partir de là, gerante() sera appelée si le
//signal identifié par nomSignal se produit
}
```

Association Signal et Fonction

- **associer** se dit sigaction() en C.
- Voici la Signature :

```
#include <signal.h>
int sigaction(int signum, //identité du signal
const struct sigaction *act, //cf. ci-dessous
struct sigaction *oldact) ;//NULL souvent
```

- Avec la définition :

```
struct sigaction {
void (*sa_handler)(int) ; //gérante
sigset_t sa_mask ; //masque à appliquer
int sa_flags ; //options
}
```

- *gérante* ci-dessus est un pointeur sur fonction. Noter qu'en C, le *nom* d'une fonction est un pointeur sur son code.

Les types de signaux

- Il existe un nombre NSIG de signaux différents, chacun étant identifié par un numéro de 1 à NSIG. NSIG est une constante définie dans le fichier <signal.h>.

Signal		Traitement par défaut
SIGHUP	Terminaison du processus leader de session	(1)
SIGINT	Frappe du car **intr** sur terminal de contrôle	(1)
SIGQUIT	Frappe du car **quit** sur terminal de contrôle	(2)
SIGILL	Détection d'une instruction illégale	(2)
SIGABRT	Terminaison provoquée en exécutant **abort**	(1)
SIGFPE	Erreur arithmétique (division par zéro, …)	(1)
SIGKILL	Signal de terminaison	(1) * non modifiable
SIGSEGV	Violation mémoire	(2)
SIGPIPE	Écriture dans un tube sans lecteur	(1)
SIGALARM	Fin de temporisation (fonction alarm)	(1)
SIGTERM	Signal de terminaison	(1)
SIGUSR1	Signal émis par un processus utilisateur	(1)
SIGUSR2	Signal émis par un processus utilisateur	(1)
SIGCHLD	Terminaison d'un fils	(3)
SIGSTOP	Signal de suspension	(4) *
SIGSUSP	Frappe du car susp sur terminal de contrôle	(4)
SIGCONT	Signal de continuation d'un signal d'un processus stoppé	(5) *

- La commande kill –l donne la liste des signaux du système
- **Traitements par défaut**: (1) terminaison de processus (2) terminaison de processus avec image mémoire (fichier core) (3) signal ignoré (sans effets) (4) suspension du processus, (5) continuation (reprise d'un processus stoppé).

Quelques Signaux (valeurs)

Signal	Valeur	Action	Signal	Valeur	Action
SIGHUP	1	T	SIGPIPE	13	T
SIGINT	2	T	SIGALRM	14	T
SIGQUIT	3	T	SIGTERM	15	T
SIGILL	4	T	SIGUSR1	30,10,16	T
SIGFPE	8	M	SIGUSR2	31,12,17	T
SIGKILL	9	TEF	SIGCHLD	20,17,18	I
SIGSEGV	11	M	SIGCONT	19,18,25	
			SIGSTOP	17,19,23	DEF

Action désigne l'action par défaut. `SIGUSR` sont des signaux sans signification particulière, laissés à la disposition de l'utilisateur.

T : terminer processus
I : ignorer signal
M : image mémoire
D : interrompre processus
E : ne peut être géré
F : ne peut être ignoré

Exemple

- Gestion du signal SIGSEGV de sortie de l'espace mémoire.

```
//déclaration de la gerante
void gestionsig (int elSig){
cout <<"recu le signal "<<elSig<<" ; " ;
cout <<"que faire que faire ?" <<endl ;
exit(1) ;
}
int main(){
struct sigaction actshoum ;
actshoum.sa_handler = gestionsig ;
int ru=sigaction(SIGSEGV, &actshoum, NULL) ;
int *demer ; demer=NULL ;
cout <<"oouupsnn" <<*demer<<"trop tard"<<endl ;
}
```

Remarques

- Un seul bit par signal est utilisé dans la table des processus du système) des signaux peuvent être perdus, par exemple dans le cas d'expéditions en rafale.

- Pour ignorer un signal le gérant s'appelle SIG_IGN : dans l'exemple, actshoum.sa_handler=SIG_IGN.
- De même, SIG_DFL désgine le gérant par défaut.

- Il faut refaire l'association (signal < - > gérant) à chaque changement de gérant.

Envoi d'un signal

- Les processus peuvent communiquer par le biais des signaux. La commande killenvoie un signal à un processus (ou groupe de processus) :

  int kill(pid_t pid, int sig);

- *sig* est le nom d'un signal ou un entier entre 1 et NSIG.

- On peut utiliser *sig*= 0 pour tester l'existence d'un processus

Valeur de *pid*	Signification
> 0	Processus d'identité *pid*
0	Tous les processus dans le même groupe que le processus
< -1	Tous les processus du groupe |*pid*|

Exemple d'envoi de signal

```
#include<sys/types.h>
#include<unistd.h>
#include<stdlib.h>
#include<stdio.h>
#include<sys/wait.h>
#include<signal.h>
voidmain(void) {
      pid_tp;
      intetat;
      if ((p=fork()) == 0) { /* processus fils qui boucle */
            while(1);
            exit(2);
      }
      /* processus pere*/
      sleep(2);
      printf("envoi de SIGUSR1 au fils %d\n", p);
      kill(p, SIGUSR1);
      // bloque l'appelant (pere) et selectionnele fils
      p = waitpid(p, &etat, 0);
      printf("etatdu fils %d : %d\n", p, etat>> 8);
}
```

Traitements par défaut

- A chaque type de signal est associé un handler par défaut appelé **SIG_DFL**
- Les traitements par défaut sont:
  - terminaison du processus (avec/sans image mémoire : fichier core)
  - signal ignoré (sans effet)
  - suspension du processus
  - continuation : reprise d'un processus stoppé.
- Un processus peut ignorer un signal en lui associant le handler **SIG_IGN**
- Les signaux **SIGKILL, SIGCONT et SIGSTOP** ne peuvent avoir que le handler **SIG_DFL**

Traitements spécifiques

- Les signaux (autres que SIGKILL, SIGCONTet SIGSTOP) peuvent avoir un handlerspécifique installé par un processus:

- La primitive **signal()** fait partie du standard de C et non de la norme de POSIX:

```
#include<signal.h>
void (*signal (int sig, void (*p_handler)(int))) (int);
```

- Elle installe le handler spécifié par *p_handler* pour le signal *sig*. La valeur retournée est un pointeur sur la valeur ancienne du *handler*.

Exemple d'installation de nouveau handler

- Signal généré par des touches du clavier (CTL/C= SIGINT)

```
#include <stdio.h>
#include <signal.h>

void hand(int signum)
{
   printf("appui sur Ctrl-C\n");
   printf("Arret au prochain coup\n");
   signal(SIGINT, SIG_DFL);
}

int main(void)
{
   signal(SIGINT, hand);
   for (;;) { }
   return 0;
}
```

Attente d'un signal

- La primitive ***pause()*** bloque en attente le processus appelant jusqu'à l'arrivée d'un signal.

```
#include<unistd.h>
intpause (void);
```

- A la prise en compte d'un signal, le processus peut :
  - se terminer car le handler associé est SIG_DFL;
  - passer à l'état stoppé; à son réveil, il exécutera de nouveau pause()et s'il a été réveillé par SIGCONT ou SIGTERM avec le handler SIG_IGN, il se mettra de nouveau en attente d'arrivée d'un signal;
  - exécuter le handler correspondant au signal intercepté.

- ***pause()*** ne permet pas d'attendre un signal de type donné ni de connaître le nom du signal qui l'a réveillé

Exemple avec SIGPIPE

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void    main() {
   int tube[2];
   int s=34;
   pipe(tube);

   //ecriture sans lecteur
   close (tube[0]);
   write (tube[1], &s, sizeof(s)); // signal SIGPIPE genere
   close (tube[1]);
   printf("ecriture terminee\n");

}
```

Exemple avec SIGPIPE avec handler

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void hand (int s) { printf("signal num: %d - ecriture non
   autorisee car pas de lecteur\n", s);
}
void   main() {
   int tube[2];
   int s=34;
   pipe(tube);
   signal (SIGPIPE, hand);
   close (tube[0]);
   write (tube[1], &s, sizeof(s)); // signal SIGPIPE genere
   close (tube[1]);
   printf("ecriture terminee\n");
}
```

Déroutement d'un signal: SIGFPE

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void Hand_sigfpe() {
   printf("\nErreur division par 0 !\n");
   exit(1);
}

main() {
        int a, b, Resultat;

        signal(SIGFPE, Hand_sigfpe);
        printf("Taper a : "); scanf("%d", &a);
        printf("Taper b : "); scanf("%d", &b);
        Resultat = a/b;
        printf("La division de a par b = %d\n", Resultat);
}
```

TP en C

TPs

- Programmer un processus qui masque les signaux SIGINT et SIGQUIT pendant 30 secondes puis rétablit le traitement par défaut de ces signaux
- Lancer un processus qui crée un fils avec lequel il communique à l'aide du signal SIGUSR1. Les 2 processus utilisent un fichier toto qui est initialement vide. Le fils y écrit les lettres minuscules de a à z, le père les majuscules A à Z de sorte que le fichier contient à la fin du traitement la chaine de caractères:
  - aAbcBCdefDEF…vwxyzVWXYZ
- Masqauge, demasquage d'un ensemble de signaux et comment connaître les signaux en attente
- Programmer un processus qui ignore les 5 premiers SIGINT et réagit au 6e
- Signal SIGSEGV et violation de mémoire
- Mise en place d'un timeout
- Utilisation de l'horloge temps réel