

Traitement du signal en temps-réel

Thomas Hueber, CNRS/GIPSA-lab

(MAJ novembre 2018)

C'est qui le prof ?

- Thomas Hueber, Ph.D
- Chercheur au CNRS, affilié au GIPSA-lab
- Contact :
 - thomas.hueber@gipsa-lab.fr
 - Twitter: @thomashueber
- Activités de recherche :
 - Technologies vocales pour l'aide au handicap (suppléance vocale, rééducation orthophoniques)
 - Traitement du signal multimodal
 - Modélisation par apprentissage
 - Plus d'informations sur : www.gipsa-lab.fr/~thomas.hueber/
- Activités d'enseignements :
 - Master SIGMA (Bayesian modeling)
 - TSTR - Phelma



Signal
Technologies
Sciences
Parole
Information
Traitement
Temps-réel
Cognition
Communication
personnes
implémentation
Parole trouble
information
Cognition
Ultraspeech
recherche
parole
aide
silencieuse
rééducation
Docteur/Ingénieur
CNRS
Lauréat
GIPSA-lab
visuel
assistance
laryngectomisées
Prix
articulaire
Électronique
technologies
systèmes
interfaces
projets
GIPSA-lab
orthophonique
vocales
Technologies
Sciences
Parole
Information
Traitement
Temps-réel
Cognition
Communication
personnes
implémentation
Parole trouble
information
Cognition
Ultraspeech
recherche
parole
aide
silencieuse
rééducation
Docteur/Ingénieur
CNRS
Lauréat
GIPSA-lab
visuel
assistance
laryngectomisées
Prix
articulaire
Électronique
technologies
systèmes
interfaces
projets

Organisation du cours

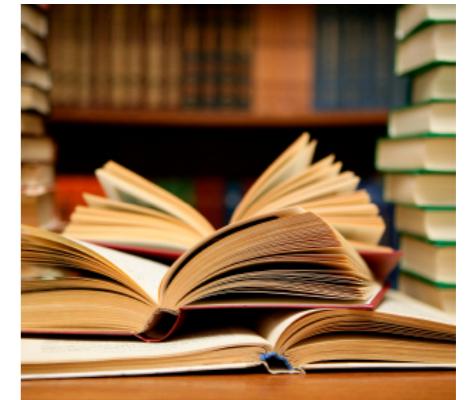
- Slides disponibles sur http://www.gipsa-lab.fr/~thomas.hueber/cours/hueber_tstr_v1_2.pdf
- Cours magistral: 2x2h (dernière heure réservée à la prise en main de l'API rtAudio)
- BE: 4 séances de 4h (dernière heure réservée à l'évaluation)
 - Travail en binôme
 - Effet audio temps-réel
 - Reverb à convolution
 - Compresseur/Limiter
 - Outils: Simulation offline en Matlab & implémentation en temps réel en C/C++



Objectifs de ce cours

- Comprendre la notion de « systèmes temps-réel » (RT)
 - Connaitre les principaux modèles théoriques de conception d'un système temps-réel
 - Prendre conscience des choix à faire lors de la conception d'un système temps-réel (matériel, système d'exploitation, language de programmation, technique d'implémentation)
- Appréhender les différents aspects liés au développement d'une application audio temps-réel sur PC/*smartphone*
- L'audio sur PC/*smartphone*
 - Comprendre l'audio sur PC (quelques mots sur les différentes API de développement: DirectX, CoreAudio, ASIO, etc.)
 - Notion de traitement par *buffer*
 - Connaitre les différents outils utilisables pour la réalisation d'un prototype (API audio RTAudio, PortAudio, environnement temps-réel de type Max/MSP, PureData, outils de simulation temps-réel de type Matlab DSP system toolbox)
- **Mettre les mains dans le cambouis !**

Références



- A. Burns and A. Wellings: “Real-Time Systems and Programming Languages”, Addison-Wesley, 4:th edition, 2009
- Real-time systems: cours en ligne de l'université de Gothenburg, Professor Jan Jonsson
- Cycle de séminaires IRCAM, MuTant, <http://repmus.ircam.fr/mutant/rtmseminars>
 - Présentation de Christoph Kirsh: Principles of real-time programming
 - Présentation de Roger Dannenberg: Principles for effective real-time music processing systems
- Jack Stankovic, “Misconceptions of Real-Time Computing”, IEEE Computer, 21(10): 10--19, 1988 - <https://www.ece.cmu.edu/~ece749/docs/Misconceptions-Stankovic.pdf>
- http://richard.grisel.free.fr/Master_OSM/5_rts-intro2_fr.pdf
- http://beru.univ-brest.fr/~singhoff/ENS/UE_Appli_Info/CM/intro.pdf

Vous avez dit « temps-réel »

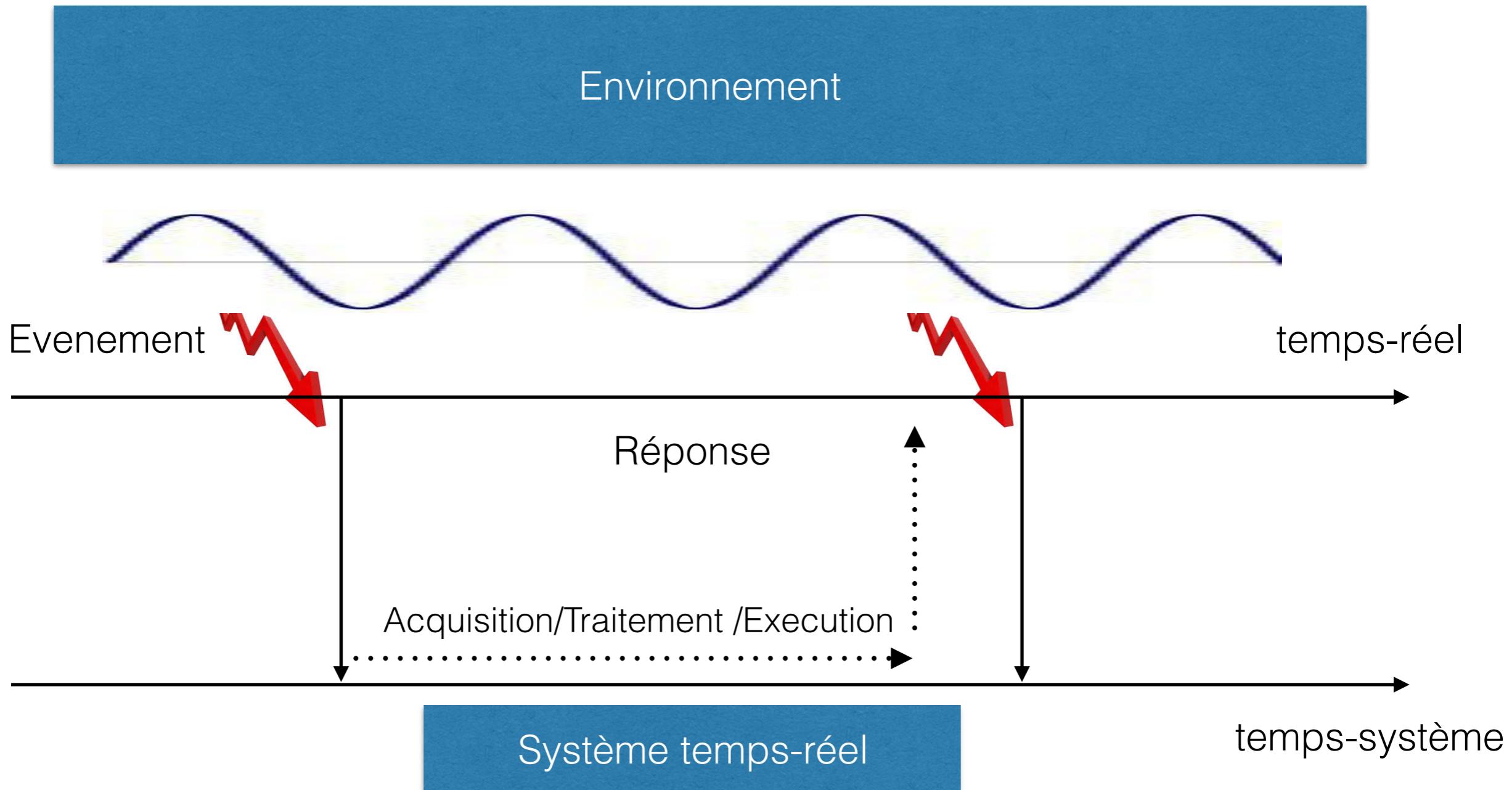
- Discutons en ! (5mn !)



Systèmes temps-réel - Définition(s) et modèles théoriques

Système temps-réel: définition(s)

- Plusieurs définitions possibles:
- System which « *controls an environment by receiving data, processing them, and returning the results sufficiently quickly to affect the environment at that time* » from Martin, James (1965). Programming Real-time Computer Systems. Englewood Cliffs, NJ: Prentice-Hall Inc. p. 4. ISBN 013-730507-9.
- Système capable de contrôler ou piloter un procédé physique à une vitesse adaptée à l'évolution de ce procédé
- System for which « *the correctness [...] depends **not only** on the logical result of computation, but also on the time at which the results are generated* » from Jack Stankovic, “Misconceptions of Real-Time Computing”, IEEE Computer, 21(10):10--19, 1988 - Head of the ‘real-Time and Embedded Systems Laboratory’ (University of Virginia engineering)

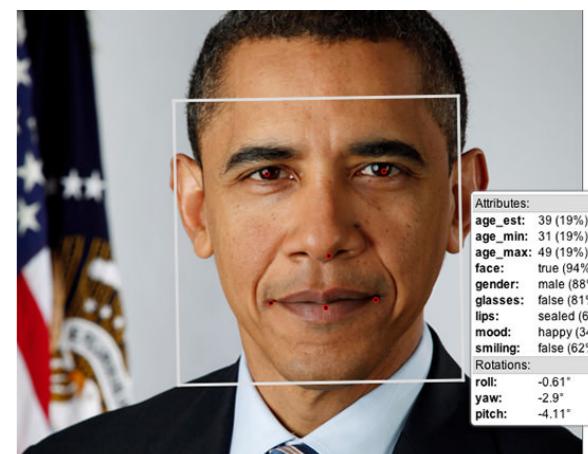
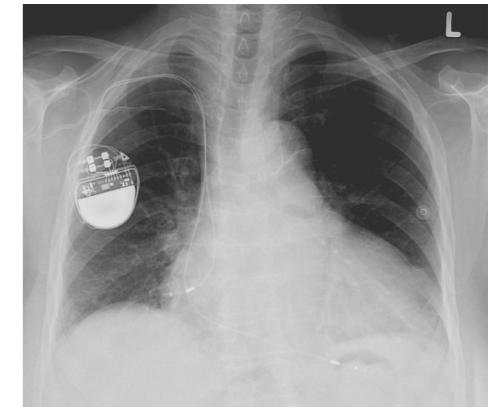


- Comportement valide si le résultat d'un traitement (notion de *correctness*) :
 - est conforme à celui attendu (voir identique à celui qui aurait été obtenu sans la contrainte d'execution en temps-réel)
 - **est rendu disponible avant une date limite (deadline)**
 - Notion de Jitter dans les systèmes temps-réel (variation du temps de réponse (<deadline))

Classement des systèmes RT selon le respect des contraintes temporelles

« Almost all computer systems of the future will utilize real-time scientific principles and technology »
(Jack Stankovic)

- *Hard real-time systems* : ne pas respecter une *deadline* équivaut à un dysfonctionnement majeur ou empêche le système de continuer de fonctionner
 - Exemples : Système ABS et air-bag d'une voiture, défibrillateur implanté, surveillance d'un réacteur nucléaire, etc.
 - Notion de *mission/safety-critical*
 - Fortes interactions « bas-niveaux » avec les composants matériels - systèmes embarqués (surveillance)
- *Soft real-time systems* : l'utilité du résultat d'un traitement diminue lorsque qu'on s'écarte de la *deadline*, mais le système continue de fonctionner (avec une dégradation de qualité)
 - Exemples : nombreuses applications multimédia (décodeur MPEG sur un lecteur de DVD, téléphonie sur IP, encodage/décodage/streaming de musique, réalité virtuelle, stabilisation video, face tracking, jeux video, etc.)
 - Implémentation envisageable sur des plateformes a priori non dédiées au temps-réel (PC/smartphone)
 - **C'est dans ce cadre dans lequel se place ce cours**



Un exemple de *soft real-time system*

Réalité augmentée



<http://augmenteddev.com>

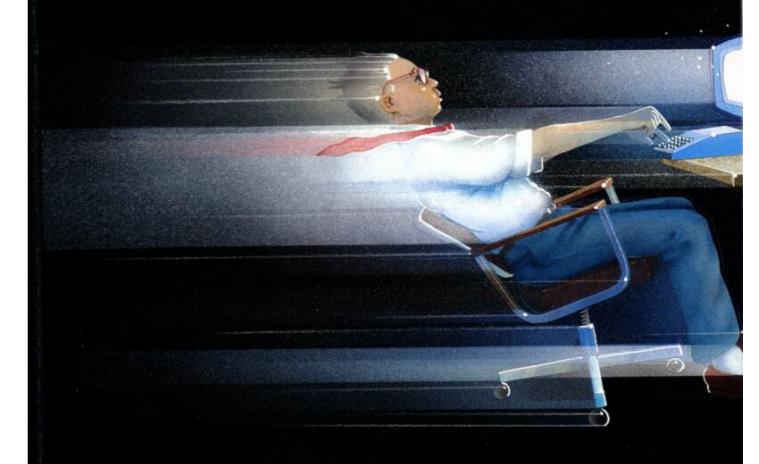
Deadline fixée par la résolution temporelle du flux video
+
latence acceptable pour l'utilisateur

(*soft real-time system*)

... dans le domaine de l'audio

- Pratiquement toutes les applications audio se voient aujourd'hui imposer une contrainte d'exécution en temps-réel
- Téléphonie/Visioconférence sur IP, débruitage actif, codage audio/video, design sonore (ex. traitement en temps-réel du son du moteur)
- Effet audio (égalisation, compression, reverberation, etc.), morphing vocal
- Synthèse sonore, instruments virtuels
- Sous-titrage et traduction temps-réel (reconnaissance/traduction/synthèse vocale)
- Contrôle du son par le geste en temps-réel
- ...

Traitement temps-réel et calcul rapide



- Execution temps-réel ne veut pas dire execution rapide !
- Exécution rapide (*fast computing*)
 - minimisation du temps nécessaire pour executer un ensemble de tâches
 - Pas de contrainte de *deadline*
 - Un super-calculateur n'est (a priori) pas un système temps-réel
- Néanmoins, les systèmes temps-réel bénéficient bien souvent des techniques permettant une execution rapide (ex: multi-core processing)
 - notamment lorsque la latence doit être inférieure à la latence perceptible par un humain (exemple : ~20 ms en audio !)

Défis posés par la conception d'un système temps-réel

- **Concurrence**

- Un système RT fonctionne en parallèle avec le monde réel (contrairement à la simulation)
- Traitement en continue de l'information (exécution « perpétuelle »).
- Le rythme des événements dépend de l'environnement !
- Système fonctionne en parallèle d'autres systèmes (ex. plusieurs applications sur un même PC)
- —> Difficulté à maintenir un comportement déterministe et reproductible (*debug* parfois compliqué)
- « **Predictabilité** » (comportement déterministe) : Peu importe la charge à un instant t , le système doit être capable de prendre une décision qui lui permettra de respecter la *deadline* imposée (ex: *critère d'arrêt d'un algorithme d'optimisation qui en converge pas*—> *solution par défaut*).

- **Efficacité**

- Les systèmes temps-réels sont souvent embarqués —> optimisation des ressources matériels (souvent limitées)

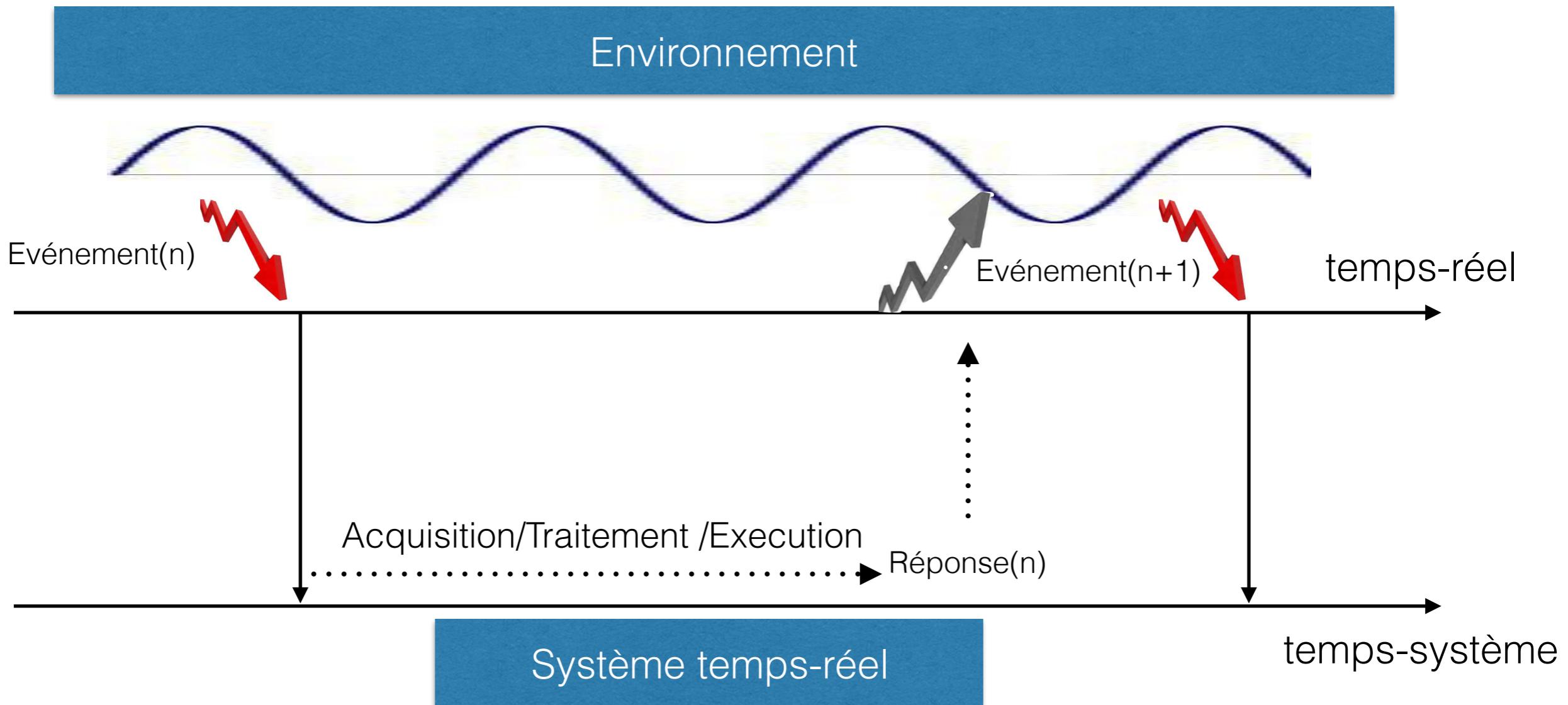
- **Tolérance** à une erreur d'exécution

- Une erreur dans le traitement d'un événement ne doit pas empêcher de traiter d'autre événement.
- Gestion fine des exceptions (*try/catch*)

- **Choix du modèle d'implémentation**

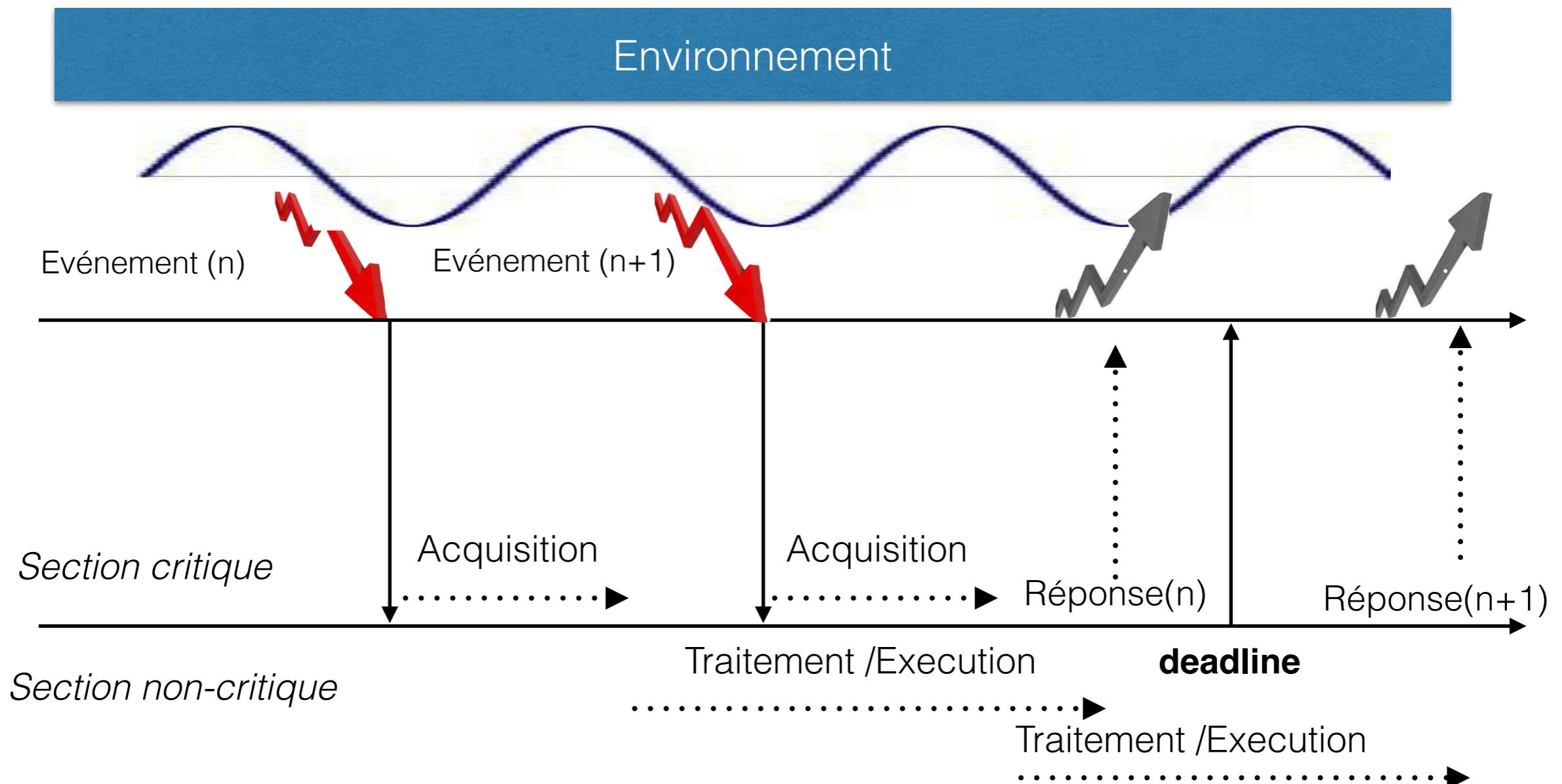
- Synchronous, scheduled, timed-programming model (Principles of real-time programming, Christoph. M. Kirsch)

Synchronous model



deadline = date du prochain événement
(Acquisition + Traitement + Execution) < temps entre 2 événements successifs

Scheduled model

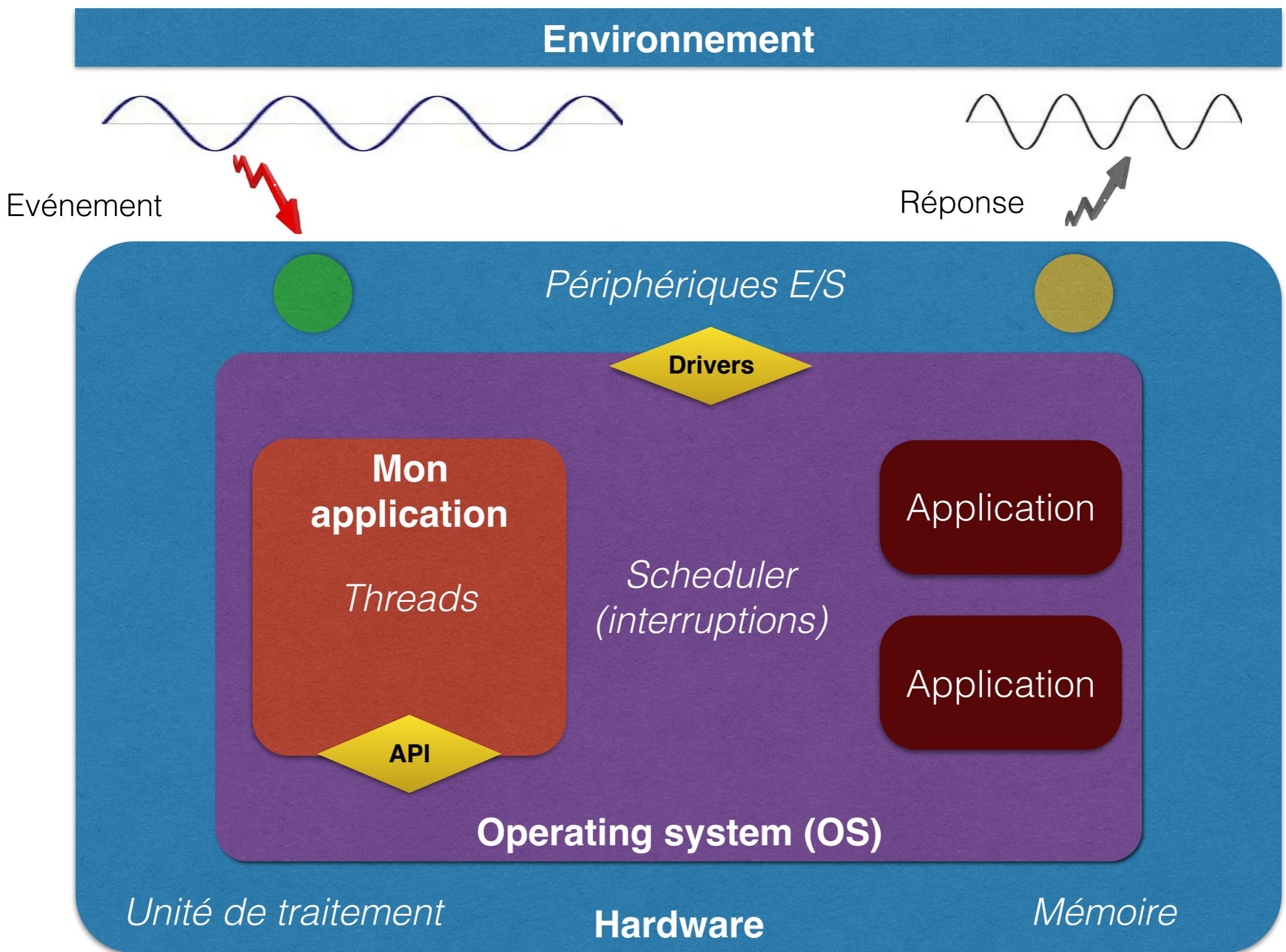


Deadlines non imposées par la cadence des événements

Priorités différentes pour l'acquisition et le traitement —> **scheduling** (*multi-task processing*)
(exemple: *video grabbing with RAM-to-disk transfer*)

Systèmes temps-réel : Conception pratique

Schéma général d'une application logicielle



Choix de l'architecture matérielle

- *Single/multiple processor (or core)*: determine le niveau de parallélisme possible
- Système complet embarqué sur une seule puce (SoC) vs. PC (smartphone, tablette)
- Ensemble des traitements sur CPU (aujourd'hui souvent multi-core) ou utilisation de processeurs dédiés :
 - **GPU** : Processeur conçu pour le traitement et la synthèse d'images (*texture mapping, shading, etc.*), mais utilisable aujourd'hui pour le traitement parallèle d'autre signaux et l'apprentissage automatique profond (deep learning) - e.g. plateforme NVIDIA CUDA
 - **DSP** (Digital Signal Processor) : Processeur spécialisé pour le traitement du signal optimisé pour le traitement rapide de données numériques en provenance d'un ADC (filtrage, FFT)
 - **FPGA** (Field Programmable Gate Array): n'est pas un « processeur » au sens où il n'exécute pas un programme stocké dans une zone mémoire dédiée; il s'agit plutôt d'un circuit composé de nombreuses cellules logiques élémentaires librement assemblables (une sorte de version « câblée en dur » de la logique associée à un algorithme).
- Aujourd'hui, de nombreux traitements audio/video complexe peuvent s'effectuer sur un CPU multi-core (ex. *face tracking*)
- Le transfert de l'execution sur DSP, GPU, ou FPGA, est souvent motivé par des contraintes de cout et de consommation d'énergie
- Choix des protocoles réseaux pour le transfert d'information et la communication inter-systèmes (TCP/IP, UDP, etc.).

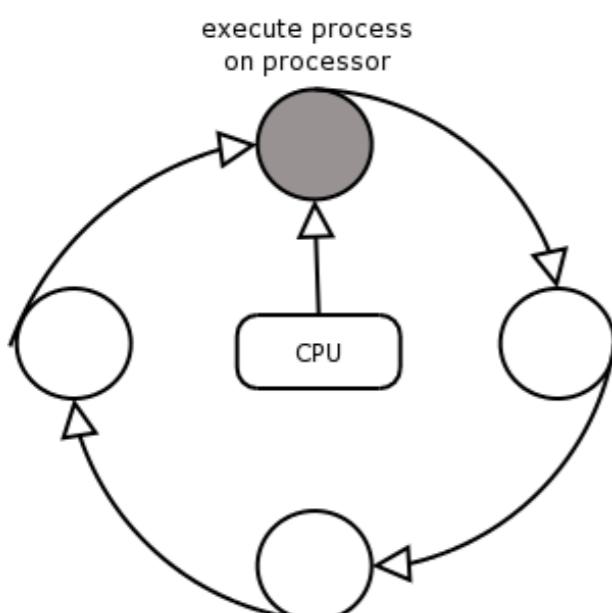
Choix du système d'exploitation

- **Rôle de l'OS (rappel) : Intermédiaire entre l'application (logicielle) et le matériel**
 - Coordonner l'exécution simultanée de plusieurs tâches utilisateurs (processus et thread)
 - Ordonnancement (*scheduling*)
 - Communication inter-processus (synchronisation par sémaphore, file de message, etc.)
 - Gestion des ressources : CPU, mémoire, périphériques
- **Real-time OS (RTOS) vs. OS standards**
 - OS standards ont pour objectif global « une exécution rapide » (sentiment de fluidité pour l'utilisateur)
 - (MS Windows, Mac OSX, iOS, Android, et la plupart des distributions Linux) ne sont pas considérés comme des OS temps-réel
 - RTOS : proposent des mécanismes pour garantir la prédictibilité (comportement déterministe) d'un processus : (LynxO, OSE, QNX, RTLinux, VxWorks, Windows CE)
 - Algorithmes d'ordonnancement avancés (*scheduling*)
 - *Minimal interrupt latency* (temps entre le moment où une interruption est déclenchée et celui où elle est effectivement traitée)
 - *Minimal thread switching latency* (latence introduite lors de la commutation de contexte)
 - Souvent utilisés pour la conception de *hard real-time systems (safety critical)* pour l'aéronautique, le militaire, etc.
 - Néanmoins, les OS non-RT modernes fournissent certains mécanismes permettant la conception de système *soft real-time* —> **adaptés pour la plupart des applications multimédia**
 - Exemple: 31 niveaux de priorités pour un thread dans Windows 7

Ordonnancement (*scheduling*)

- Algorithme d'allocation des ressources (CPU, mémoire, bande passante) pour les processus et *threads*
- Un compromis entre :
 - Débit (*throughput*) = le nombre total de processus ayant terminés leur execution par unité de temps
 - Adapté pour une execution « rapide »
 - Latence (*latency*) = temps entre le moment où une interruption est déclenchée et celui où elle est effectivement traitée
 - Adapté pour une execution « temps-réel » (respect des deadlines)
- De nombreux algorithmes, « First Come, First Served », « Fixed priority pre-emptive scheduling », « **Round-robin scheduling** », « **Multilevel queue scheduling** » etc.
- Le detail de ces algorithmes sort du cadre de ce cours (mais beaucoup de ressources en ligne !)
- A retenir :
 - les OS standards utilisent un algorithme hybride entre **Multilevel-queue + Round-robin**
 - Chaque interruption impose une commutation de contexte qui introduit un délai (context switching) !
 - **Avec un OS non temps-réel, un processus peut être interrompu (*preemptive scheduling*) sans que le programmeur ne puisse l'empêcher (totalelement) !!!**

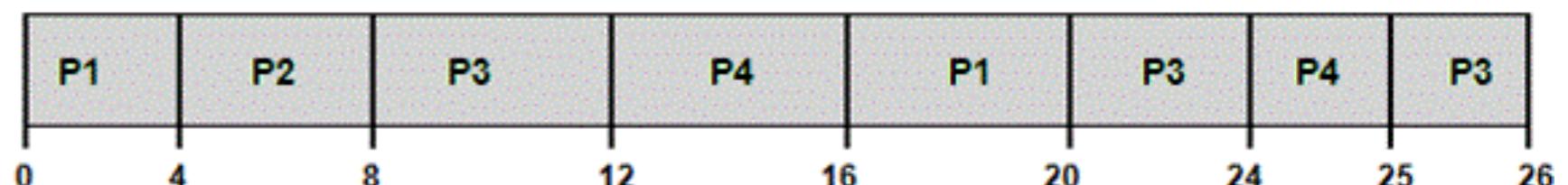
- *Round-robin* = tourniquet : Chaque processus est sur le tourniquet et passe devant le processeur pendant un temps fini (appelé quantum)



Process	Arrival	Service	
	Time	Time	Time
1	0		8
2	1		4
3	2		9
4	3		5

Note:
Example violates rules for quantum size since most processes don't finish in one quantum.

Round Robin, quantum = 4, no priority-based preemption



$$\text{Average wait} = ((20-0) + (8-1) + (26-2) + (25-3)) / 4 = 74/4 = 18.5$$

Take-home message

- Système temps-réel = respect des deadlines
- Différents modèles d'implémentation
 - Séquentielle (adaptée à un synchronous model / time-triggered system)
 - Parallèle (adaptée à un scheduled model / event-triggered system)
- *Hard real-time systems : (safety critical)*
 - Processeurs dédiés (DSP, GPU, FPGA)
 - Real-time operating system
- *Soft real-time systems* (notamment la plupart des applications multimédia)
 - Implémentation possible sur un CPU multi-core (et GPU)
 - Standard OS avec préemption non souhaitée des processus de traitement (et parfois perçue comme aléatoire par le programmeur)
 - Programmation parallèle souvent nécessaire



Traitement audio en temps-réel

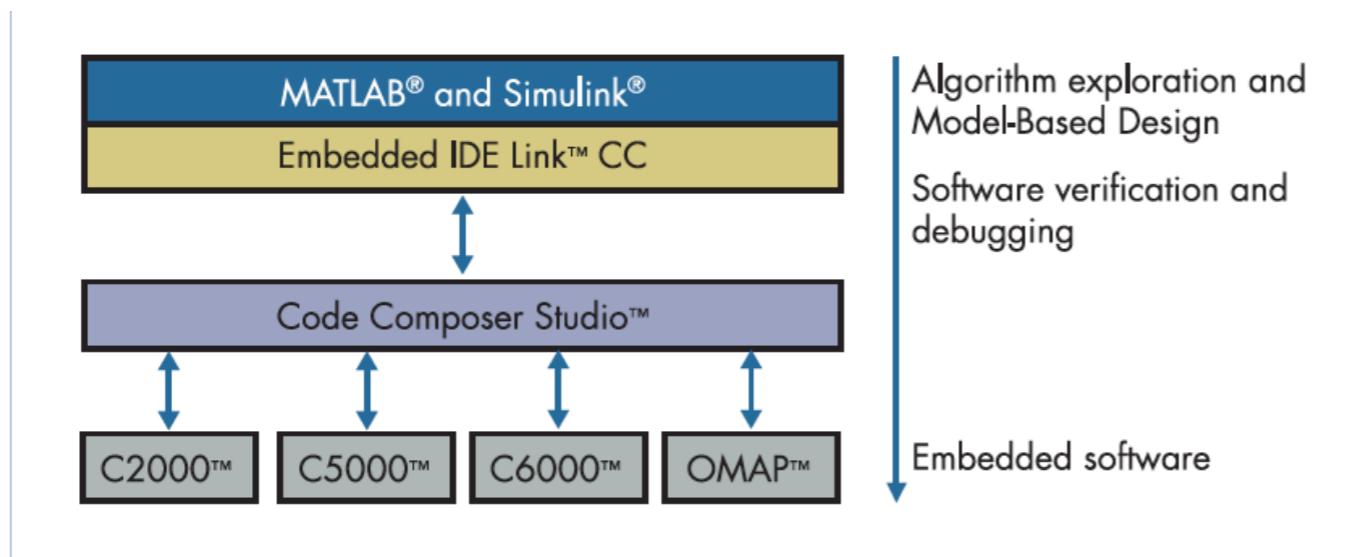
Digital Signal Processor

- Contenu « historique » du cours TSTR (jusqu'en 2013, programmation en assembleur)
- Aujourd'hui, de nombreuses applications multimédia sont conçues sans faire appel « explicitement » à ces processeurs spécialisés, notamment pour des applications destinées à une exécution sur des plateformes de type PC/tablettes/smartphone
- L'utilisation sur DSP d'une tache particulière semble être aujourd'hui réservée de plus en plus à des systèmes embarqués haute performance (*hard real-time systems*), *safety-critical* (ex. défibrillateur implanté), faible consommation (téléphones portables).
- Dans ce cours, **nous ciblons le traitement audio temps-réel sur des plateformes grand public, et tournant sur des OS standards (i.e. non temps-réel) tels que Windows, Mac OS, Linux, iOS, Android**
- Mais de nombreux concepts sont communs (ex. notion de traitement par buffers, stockage par buffers circulaires, etc.)

- **Digital Signal Processor** = microprocesseur spécialisé dédié au traitement numérique du signal en temps-réel
- Large spectre d'applications :
 - Communications numériques : téléphonie mobile, modems, fax...
 - *Consumer electronics* : audio / Hi-Fi, MP3, téléviseurs, DVD / Blue- Ray, appareils photos numériques, imprimantes, GPS, instruments de musique électroniques...
 - Médical : instrumentation, implants cochléaires, défibrillateurs, ...
- Spécificités par rapport à un autre type de processeur :
 - Instructions arithmétiques adaptées aux algorithmes de traitement du signal (ex. *instruction Multiply And Accumulate* = multiplication de 2 nombres et ajout d'un 3ème nombre en 1 seul cycle d'horloge ! très pratique pour la convolution et le FIR)
 - Prise en charge de la multiplication de nombre complexes (ex. Texas Instrument TMS320C6472)
 - Calcul en nombre entiers (plupart des DSP) ou en virgule flottante (une minorité)
 - Forte optimisation des *boucle For*
 - *Mode d'adressage spécialisés dans des situations qu'on ne trouve que dans le traitement de signal* (ex: mode d'adressage bit-reverse pour le calcul des FFT base 2)
 - *etc...*
- Programmation en assembleur ou en C
- **Permet un traitement échantillon par échantillon (latence ~ période d'échantillonnage < 10e-4 seconde = imperceptible)**

les DSP aujourd’hui

- *Code Composer Studio*, un IDE (*Integrated Development Environment*) dédié à la programmation des DSP Texas Instrument TMS320 (dérivé d’Eclipse).
 - Programmable en C ou en ASM
- Matlab peut se connecter à *Code Composer Studio* pour être utilisé comme *TestBench*



Combining MATLAB®, Simulink®, and Embedded IDE Link™ CC to provide an integrated environment for verifying, debugging, visualizing, and validating embedded software on TI DSPs.

```
/* Main Code */
main()
{
    y = dotp(a, x, 40);
}

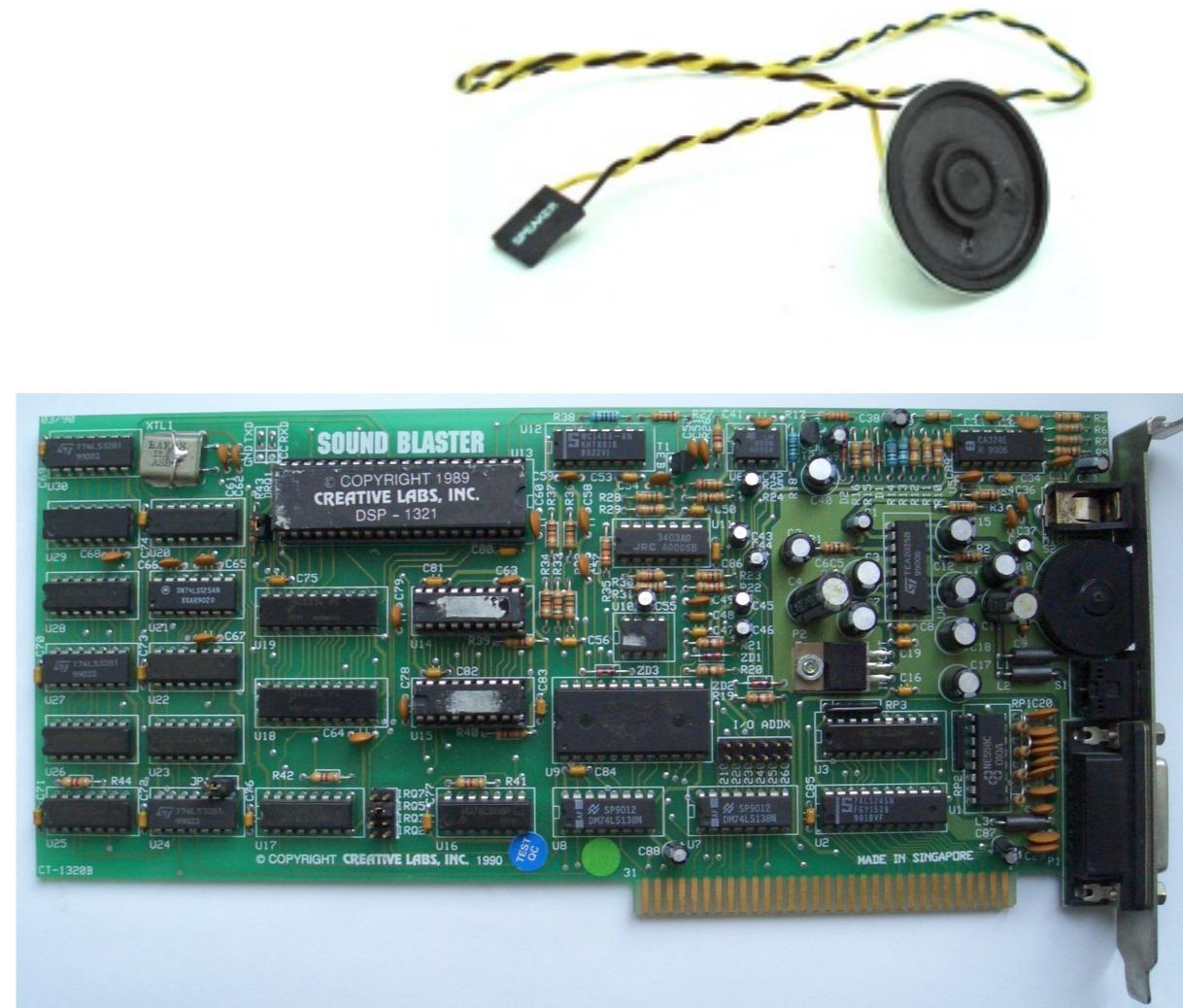
int dotp(short *m, short *n, int count)
{
    int i;
    int sum = 0;

    for (i=0; i < count; i++)
    {
        sum += m[i] * n[i];
    }
    return (sum);
}
```

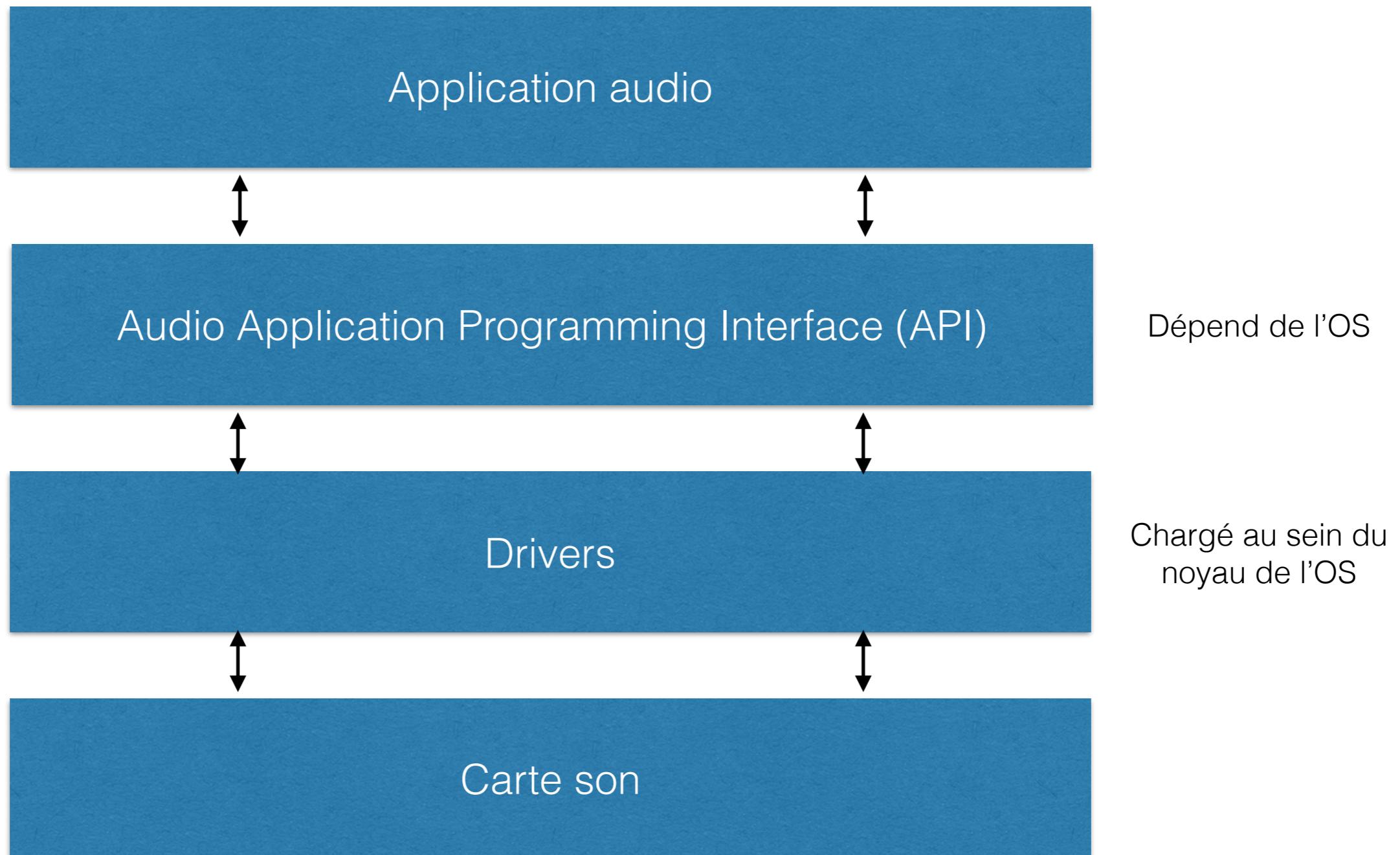
Name	Value	Type	Radix
y	11480	int	dec
x	0x00000070	short[40]	hex
a	0x00000020	short[40]	hex

L'audio sur PC : la carte son

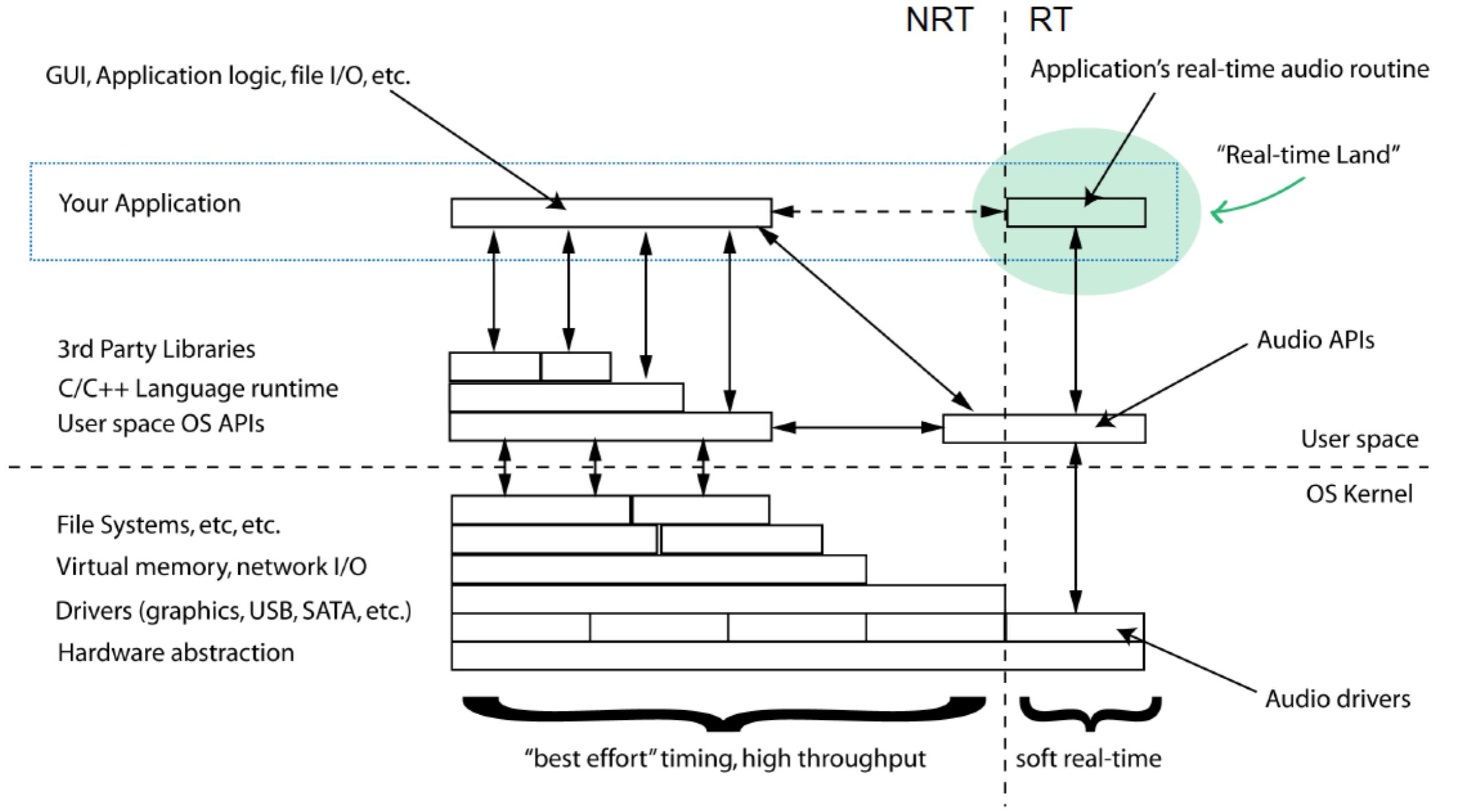
- Pour la petite Histoire :
 - 1981: le PC Speaker !
 - 1989 : Premières cartes sons grand public capables d'enregistrer (8-bit, 12 kHz)
- Paramètres du CODEC (=ADC/DAC sur un seul chip)
 - Fréquence échantillonnage possible (44100, 48000, 96000, 192 000 kHz)
 - Résolution (16 bits, 24 bits, 32 bits)
 - Taille du buffer d'échantillons (*hardware* sur certaines cartes son à usage professionnel, *software* sinon) : 32, 64, 128, ..., 4096
- Certaines cartes contiennent un ou plusieurs DSP



Du *hardware* au *software* . . .



Real-time audio programming context

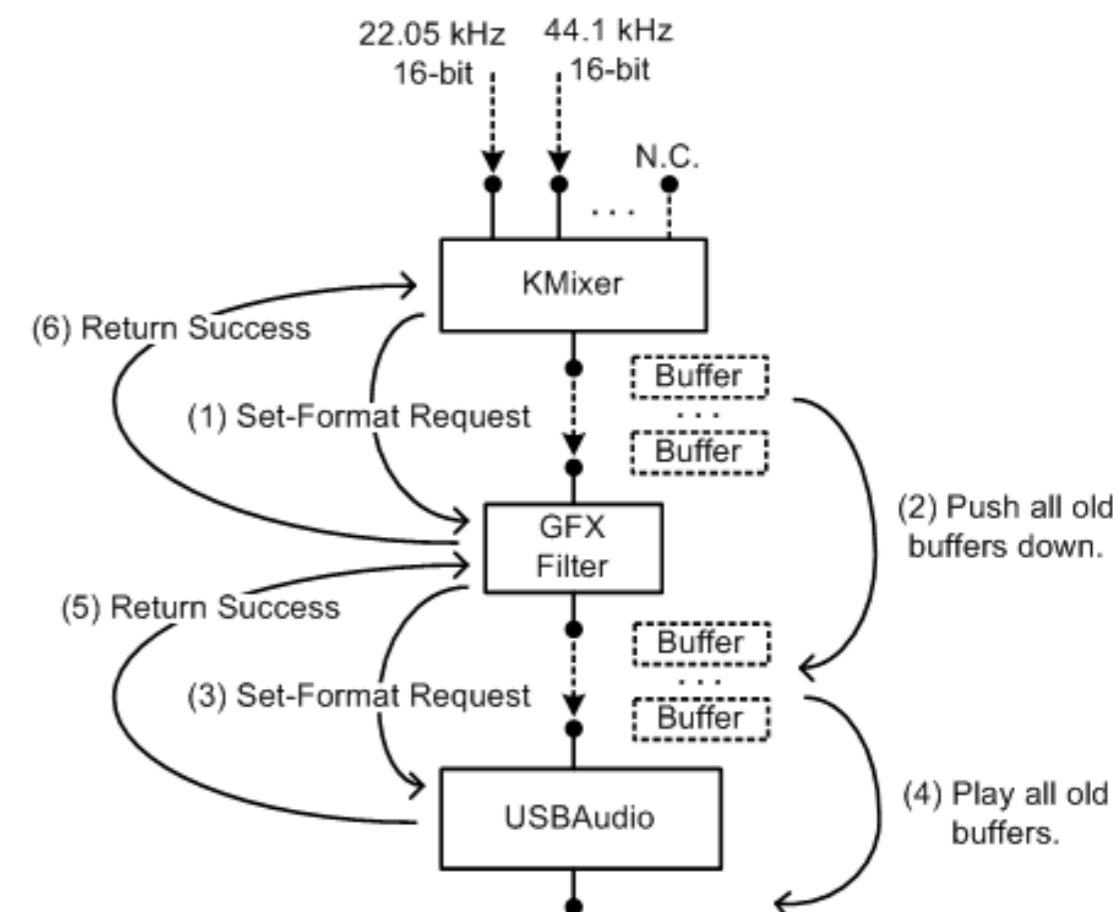


See also:

<http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing>

Du hardware au software

- Architecture multi-couche :
 - facilite le développement - programmation haut niveau (le programmeur se préoccupe peu des aspects *hardware*)
 - peut introduire des traitements intermédiaires dont **le programmeur n'a souvent pas connaissance** et qui peuvent perturber le bon fonctionnement de l'application
 - Exemple : KMixer (Kernel Audio Mixer) Windows 98 à XP dédié au :
 - Mixage des flux audio multiples au sein du système
 - **Re-échantillonnage**, re-quantification
 - Routage
 - Permet (par exemple) de travailler a priori à n'importe quelle fréquence d'échantillonnage, peu importe les caractéristiques des ADC/DAC
 - Forte latence



• Permet (par exemple) de travailler a priori à n'importe quelle fréquence d'échantillonnage, peu importe les caractéristiques des ADC/DAC



• Forte latence

API audio

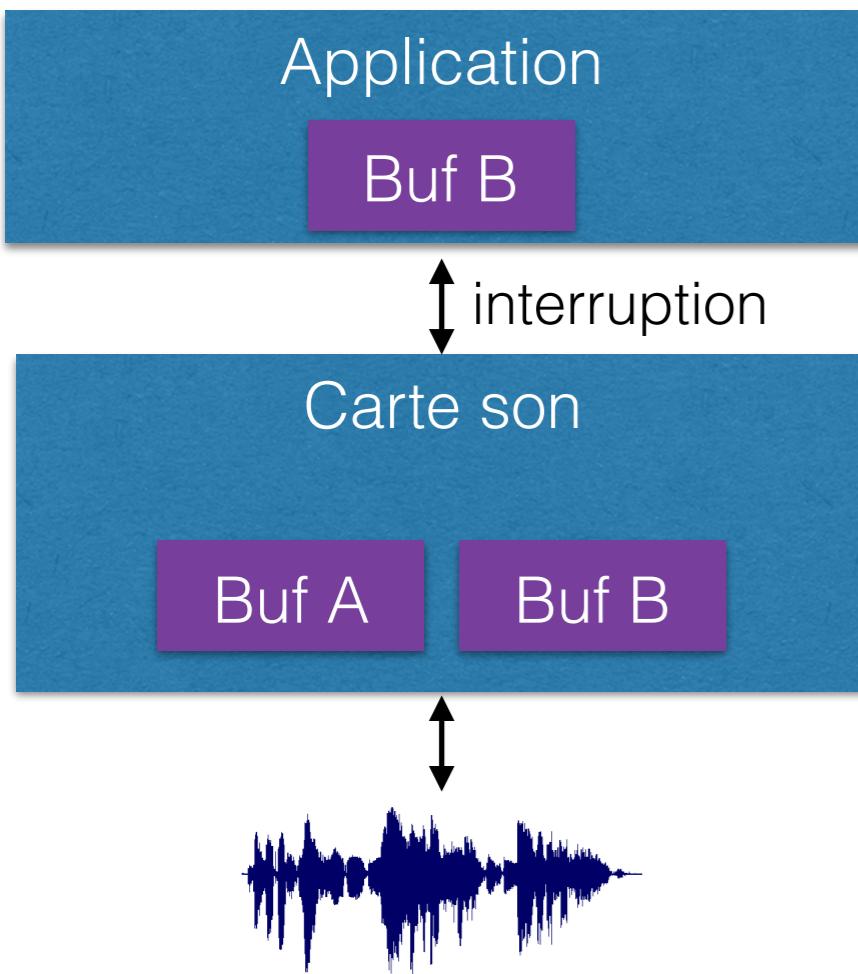
- API : *Application Programming Interface* = interface de programmation
 - ensemble normalisé de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels
 - « briques de fonctionnalités » fournies par des logiciels tiers
- API audio liée à un OS :
 - API Microsoft
 - Windows 98-XP : WDM, MME, DirectSound
 - Windows Vista-8 : WASAPI (refonte complète du système audio :-)
 - Windows 10 : Audiograph
 - **Des API dédiées au traitement audio professionnel** : Protools HD, **ASIO**
 - Mac OSX : CoreAudio
 - Linux : ALSA, OSS, Jack
- API audio cross-platform : **rtAudio**, portAudio
- API multimédia (avec fonctionnalités audio) : SDL, JUCE, etc.

Un exemple: Audio Stream Input/ Output (ASIO)

- API de développement + protocole de conception hardware + drivers pour le développement d'applications audio faible latence et haute fidélité sous Windows
- Principe : limiter les couches logicielles entre le matériel et l'application pour diminuer la latence
- Notion de "*bit identical*" : les échantillons rendus accessibles au programmeur par l'intermédiaire de l'API sont identiques à ceux en sortie de l'ADC
- 2 options :
 - Utiliser une carte son compatible ASIO (toutes les cartes sons professionnelles)
 - Utiliser un utilitaire qui *bypass* le KMixer (responsable de la latence) tel que ASIO4ALL
 - rtAudio (utilisé dans les BE) peut utiliser le protocole ASIO pour la gestion de l'audio sous Windows

Traitement par blocs

- Dans une application audio sur PC, le traitement des échantillons s'effectue principalement par blocs (et non échantillons par échantillons)
- Dans la plupart des API, la carte son (i.e. le drivers) communique les échantillons audio à l'application par l'intermédiaire d'une FIFO
 - Taille \geq à 2 buffers (pour permettre l'écriture du prochain buffer pendant le traitement du buffer courant)



Exemple (API rtAudio)

```
int record( void *outputBuffer, void *inputBuffer, unsigned int nBufferFrames,
            double streamTime, RtAudioStreamStatus status, void *userData )
{
    if ( status )
        std::cout << "Stream overflow detected!" << std::endl;

    // Do something with the data in the "inputBuffer" buffer.

    return 0;
}
```

La latence dépend (notamment) de la taille des buffers



Quelques mots sur la latence

- La latence maximale autorisée dépend de l'application (téléphonie, télévision, réalité augmentée, effet/synthèse sonore, etc.)
- Perception de sa propre voix (Delayed Auditory Feedback, DAF)
 - Utilisé notamment pour le traitement du bégaiement —> diminution des disfluences si retard > 150 ms
 - Chez un sujet sain, apparition de disfluences si retard > 150 ms
- Latence dans un système de téléphonie mobile
 - *EDGE can carry a bandwidth up to 236 kbit/s (with end-to-end latency of less than 150 ms)*
 - VOIP : valeurs typiques comprises entre 20 ms et 150 ms
 - **Conversation difficile quand la latence > 200 ms**
- Latence en audio-visuel : détection d'une mauvaise synchronisation parole/mouvement des lèvres dès 25 ms de délai (< 1 trame à 25 fps)
- Latence en effet/synthèse sonore : < 10 ms pour jouer un piano (MIDI), < 5 ms pour un instrument percussif !
- Synchronisation audio via Internet (ex : musiciens repartis sur des sites distants) aujourd'hui très difficile !
 - Exemple : *ping www.louvre.fr* à partir de Grenoble —> Minimum latency ~ 40 ms (average latency ~100 ms !)
- Quelques ordres de grandeur : 5 ms à 44100 Hz ~ 220 échantillons, 5 ms à 16 kHz = 80 échantillons
512 échantillons à 44100 Hz ~11 ms

Modèle producteur-consommateur



illustration dans le cas de l'enregistrement audio

- **Over-run** : L'application ne lit pas les données assez rapidement, le buffer est écrasé avec de nouvelles données
- **Under-run** : L'application lit les données trop rapidement, le buffer ne contient pas de donnée et la lecture provoque une erreur.
- Pour que le producteur évite l'over-run —> *go to sleep* (avec risque de perte de données)
 - Réveil par le consommateur lorsque ce dernier a retiré des éléments de la FIFO, puis nouvel tentative de ré-écriture des données
- Raisonnement symétrique pour le consommateur dans le cas où la FIFO est vide
- Implémentation basée sur un mécanisme de communication inter-processus (par exemple des sémaphores)

Programmation audio temps-réel: quelques bonnes pratiques

Contexte : Implémentation d'un algorithme de traitement du signal dans le *callback* audio fourni par une API (ex: ASIO, rtAudio, etc.), qui s'exécute dans un thread séparé, géré par un OS non-temps-réel (ex: Windows, Mac OS X, etc.)

```
int record( void *outputBuffer, void *inputBuffer, unsigned int nBufferFrames,
            double streamTime, RtAudioStreamStatus status, void *userData )
{
    if ( status )
        std::cout << "Stream overflow detected!" << std::endl;

    // Do something with the data in the "inputBuffer" buffer.

    return 0;
}
```

- Chaque over/under-run risque d'introduire des artéfacts dans le signal audio traité (*glitch*)
- Sources de *glitch* :
 - *Blocking* : appel bloquant pendant le *callback* audio (déblocage d'un sémaphore, lecture/écriture d'une donnée sur le disque, attente de données en provenance du réseau)

A lire : <http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing>

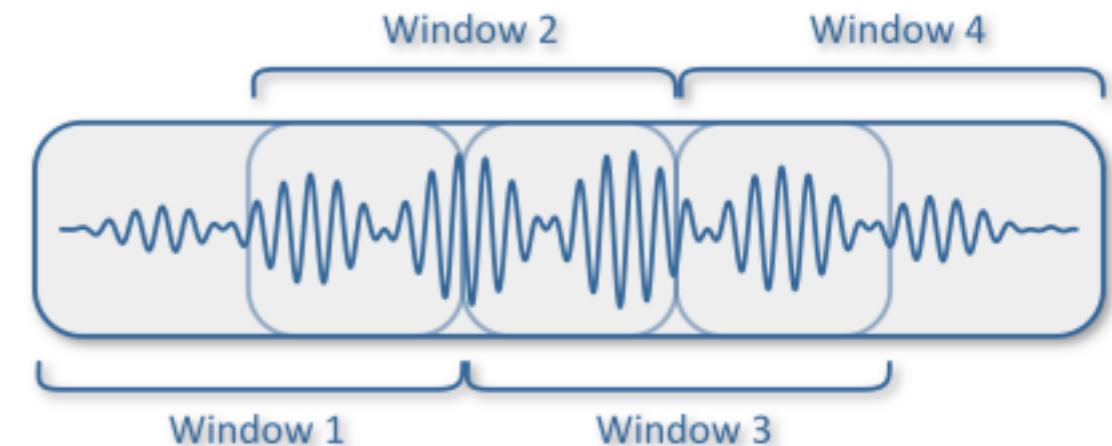
- Inversion de priorités :
 - le *thread* en charge du *callback* audio se voit généralement attribuer une priorité « haute ».
 - Les autres *threads* comme par exemple l'interface graphique (GUI) ont une priorité moyenne, elle est donc facilement interruptible.
 - Si le traitement audio possède une ressource commune avec le GUI (zone de mémoire partagée ou une section critique), pour par exemple récupérer un paramètre du traitement, alors il y a risque d'**inversion de priorité**
 - i.e. le *thread* audio (pourtant en priorité haute) est bloqué tant que le *thread* de priorité basse a la main sur la ressource commune
- Allocation de la mémoire :
 - Eviter de faire les allocations mémoire dans le *callback* audio (ex: malloc(), free() en C, ou new, delete en C++) —> attention des routines cachées dans des librairies peuvent le faire à votre insu !
 - Pourquoi ?
 - l'allocation de la mémoire peut nécessiter l'utilisation d'un mécanisme de blocage (*lock*) pour protéger de la mémoire partagée entre *thread* —> risque d'inversion de priorité
 - l'allocation de la mémoire fait appel à l'OS ... dont on ne maîtrise pas le comportement. Dans le pire des cas, l'OS peut décider de paginer la mémoire sur le disque ! (catastrophe ...)
 - Solution : pré-allouer la mémoire en dehors du callback audio ou gérer soit même l'allocation de mémoire à partir d'une grande zone pré-allouée au démarrage de l'application.

Techniques de TS en temps-réel

Stratégie de stockage des échantillons

- De nombreux traitements nécessitent le stockage de plusieurs buffers d'échantillons
 - Exemple : Filtrage FIR, codage LPC, convolution, analyse spectrale par fenêtre glissante, etc.
- Dans de nombreux cas, on utilise une structure de type « buffer circulaire »

$$y_n = \sum_{k=0}^K a_k x_{n-k}$$



Buffer circulaire (*ring buffer*)



- Structure de données très utilisée en traitement audio (et également en traitement video) temps-réel.
- Principe : un buffer de taille fixe, circulaire (« début connecté à la fin »)
 - Taille fixe : Pas de ré-allocation au cours du traitement
 - Circulaire : permet une implémentation efficace d'une FIFO (modèle producteur-consommateur)

Ecriture d'un premier élément



Ecriture de 2 autres éléments (producteur)



Lecture des 2 éléments les plus anciens (consommateur)



Remplissage total du ring buffer



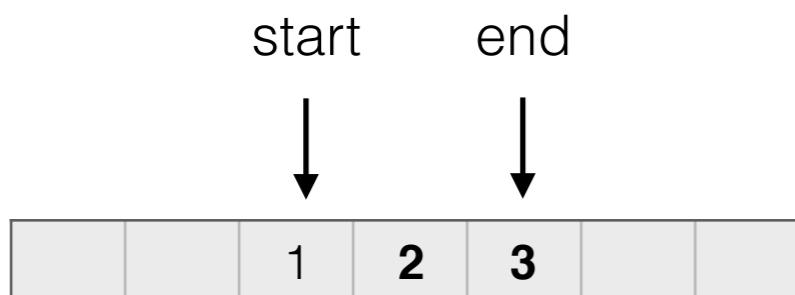
Ecrasement des données les plus anciennes (over-run)



Buffer circulaire (*ring buffer*)



- La gestion d'un buffer circulaire s'effectue généralement à l'aide de 4 variables (idéalement des pointeurs si le langage de programmation le permet)
 - la taille du buffer
 - ex : int BufSize = 1024;
 - Un pointeur vers la zone mémoire utilisée pour le stockage des échantillons
 - ex: *double *Buf=calloc(BufSize,sizeof(double))*
 - Une tête de lecture pointant vers la première donnée valide pour le consommateur
 - Une tête d'écriture pointant vers la dernière donnée écrite (par le producteur)



Convolution (filtrage linéaire)

- Rappel : Convolution discrète de 2 signaux discrets (à support finis) x et h de taille respective $K+1$ et $M+1 \rightarrow$ signal y de taille $K+M+1$ ($K+1+M+1-1$)

$$y[n] = \sum_{k=-\infty}^{\infty} x[k] h[n-k].$$

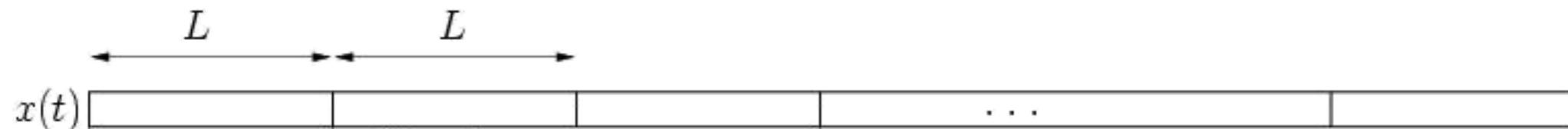
$$y[n] = \sum_{k=\max(n-M,0)}^{\min(n,K)} x[k] h[n-k].$$

$$\begin{aligned} y[0] &= x[0] h[0] \\ y[1] &= x[0] h[1] + x[1] h[0] \\ y[2] &= x[0] h[2] + x[1] h[1] + x[2] h[0] \\ &\vdots \quad \vdots \quad \vdots \\ y[M] &= x[0] h[M] + x[1] h[M-1] + \cdots + x[M] h[0] \\ y[M+1] &= x[1] h[M] + x[2] h[M-1] + \cdots + x[M+1] h[0] \\ &\vdots \quad \vdots \quad \vdots \\ y[K] &= x[K-M] h[M] + \cdots + x[K] h[0] \\ y[K+1] &= x[K+1-M] h[M] + \cdots + x[K] h[1] \\ &\vdots \quad \vdots \quad \vdots \\ y[K+M-1] &= x[K-1] h[M] + x[K] h[M-1] \\ y[K+M] &= x[K] h[M]. \end{aligned}$$

En temps-réel, x n'est pas connu entièrement !!

Solution : convolution par bloc

Convolution par blocs



Un bloc/*buffer* x_k de L échantillons peut s'écrire :

$$x_k[n] \stackrel{\text{def}}{=} \begin{cases} x[n + kL] & n = 1, 2, \dots, L \\ 0 & \text{otherwise,} \end{cases}$$

La convolution du signal complet x s'écrit comme la somme de convolution à court-terme de chaque bloc x_k par h :

$$\begin{aligned} y[n] &= \left(\sum_k x_k[n - kL] \right) * h[n] = \sum_k (x_k[n - kL] * h[n]) \\ &= \sum_k y_k[n - kL], \end{aligned}$$

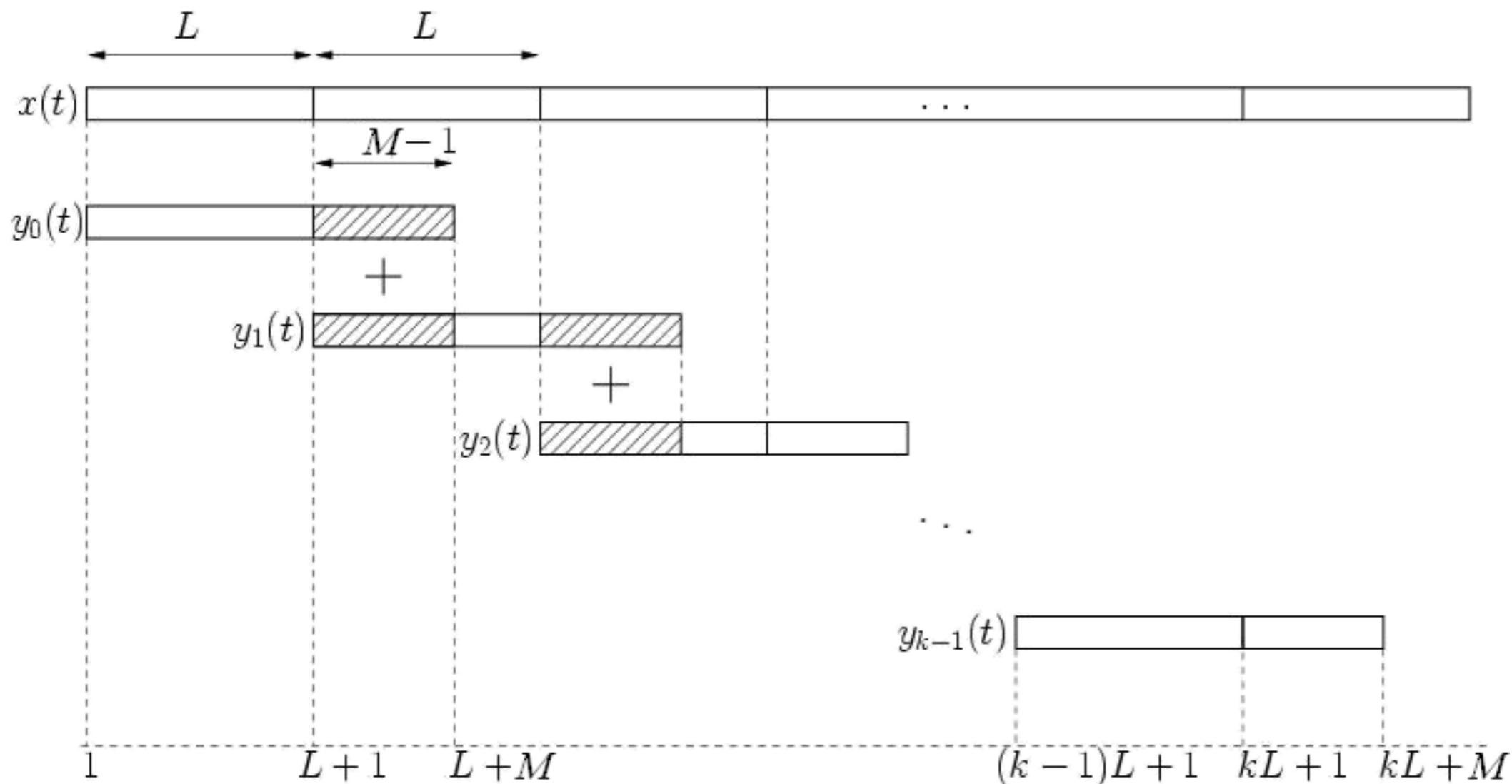
$$y_k[n] \stackrel{\text{def}}{=} x_k[n] * h[n]$$

| ← Taille $L+M-1$!!

→

overlap-add

Overlap-add



- Nécessite de stocker le dernier *buffer* traité
- Latence = 1 *buffer* + temps de calcul

Overlap-add

- Convolution dans le domaine temporel ou dans le domaine fréquentiel pour le traitement de chaque buffer
 - Domaine temporel : $y_k[n] = \sum x_k[k] h[n - k]$.
 - très lourd et potentiellement intractable dans le cas d'une réponse impulsionnelle « longue » (cf BE)
 - Domaine fréquentiel : $y_k[n] = \text{IFFT}(\text{FFT}(x_k[n]) \cdot \text{FFT}(h[n]))$
 - Attention: nombre de points de FFT N > L+M-1 (zero-padding de x et h —> équivalent à une convolution circulaire)
 - beaucoup plus rapide ! (en temps-réel, pré-calcul de FFT(h[n]) parfois possible)

- Presentation du BE

Projet 1 : Effet audio de type *Reverb à convolution* en temps-réel

- Effet audio de spatialisation visant à créer l'impression d'une écoute dans un lieu plus ou moins vaste.
- Très utilisé dans la production musicale pour le traitement des instruments et des voix
- Simule le phénomène de reverberation du son : persistence du son émis dans un lieu, après que l'émission de ce son ait cessée.
 - Cette persistence est due aux multiples reflexions du son dans un milieu physique.
 - Ces reflexions décroissent progressivement notamment en raison des phénomènes d'absorption.

(jouer le son)

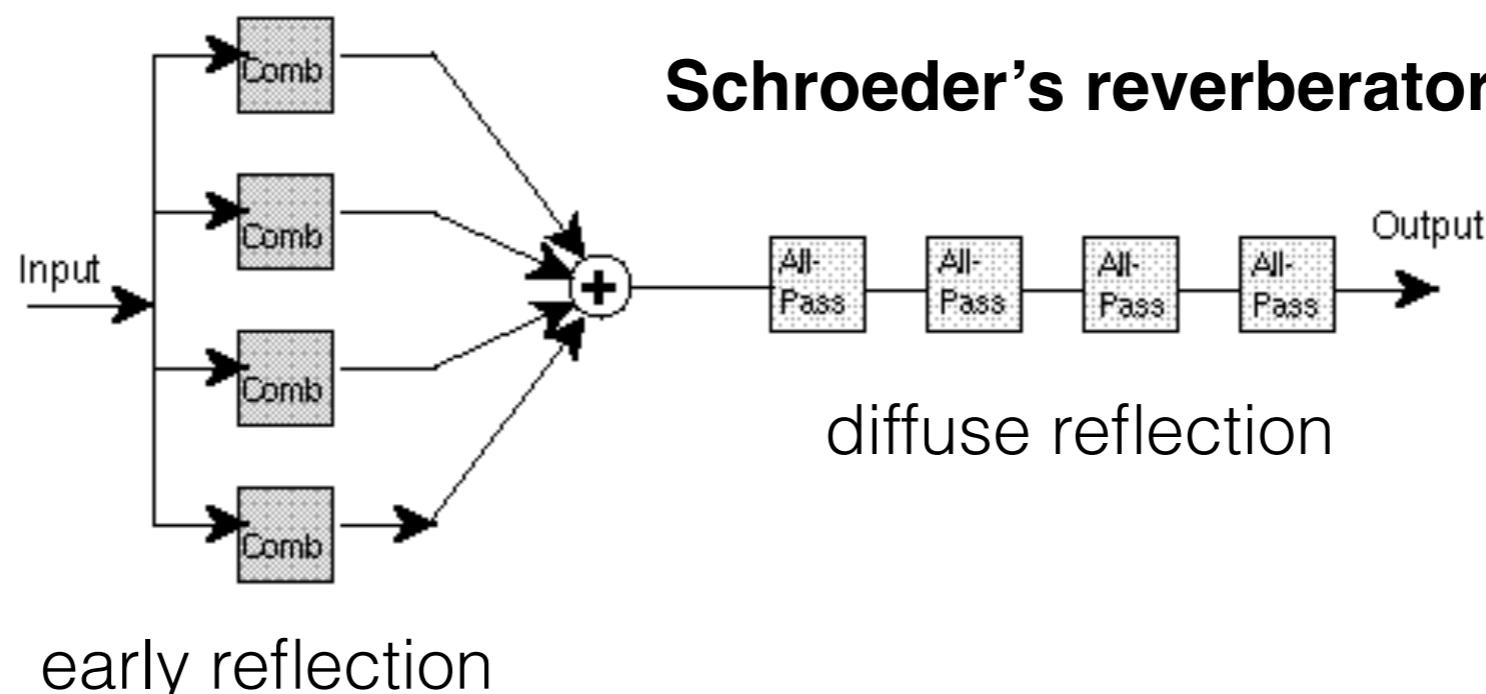
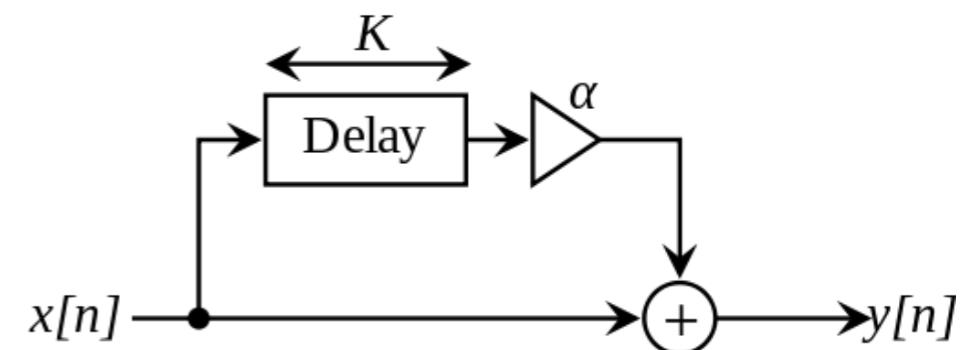
Effet Reverb

- Systèmes basés sur des lignes de retard

- Utilisation de plusieurs *Comb filter* en parallèle: ajout à un signal d'une version retardé de lui-même (simulation des reflexions)

Comb filter

$$y[n] = x[n] + \alpha x[n - K]$$



Schroeder's reverberator

diffuse reflection

Reverb à convolution

- Effet de reverberation utilisant un enregistrement de la réponse impulsionnelle de l'espace que l'on cherche à simuler
- **Convolution** (en temps-réel !) du signal à traiter avec la réponse impulsionnelle
- Attention, cette réponse impulsionnelle peut être de plusieurs secondes !
- Très haute qualité (dépend de la qualité de l'enregistrement de la réponse impulsionnelle).
- Quelques exemples de réponses impulsionnelles :

vous pouvez enregistrer les vôtres ...

Objectifs du projet

- Prise en main d'un API standard de traitement du signal (audio) en temps-réel
- Implémentation de la convolution en temps-réel d'un signal audio (microphone)
 - Domaine temporel / fréquentiel
 - Implémenter l'effet reverb en C/C++ sous UNIX à l'aide de l'API rtAudio (une implementation sous Windows est aussi possible)
- Evaluation de la latence globale E/S (en fonction de la taille et du nombre de buffers interne, etc.)

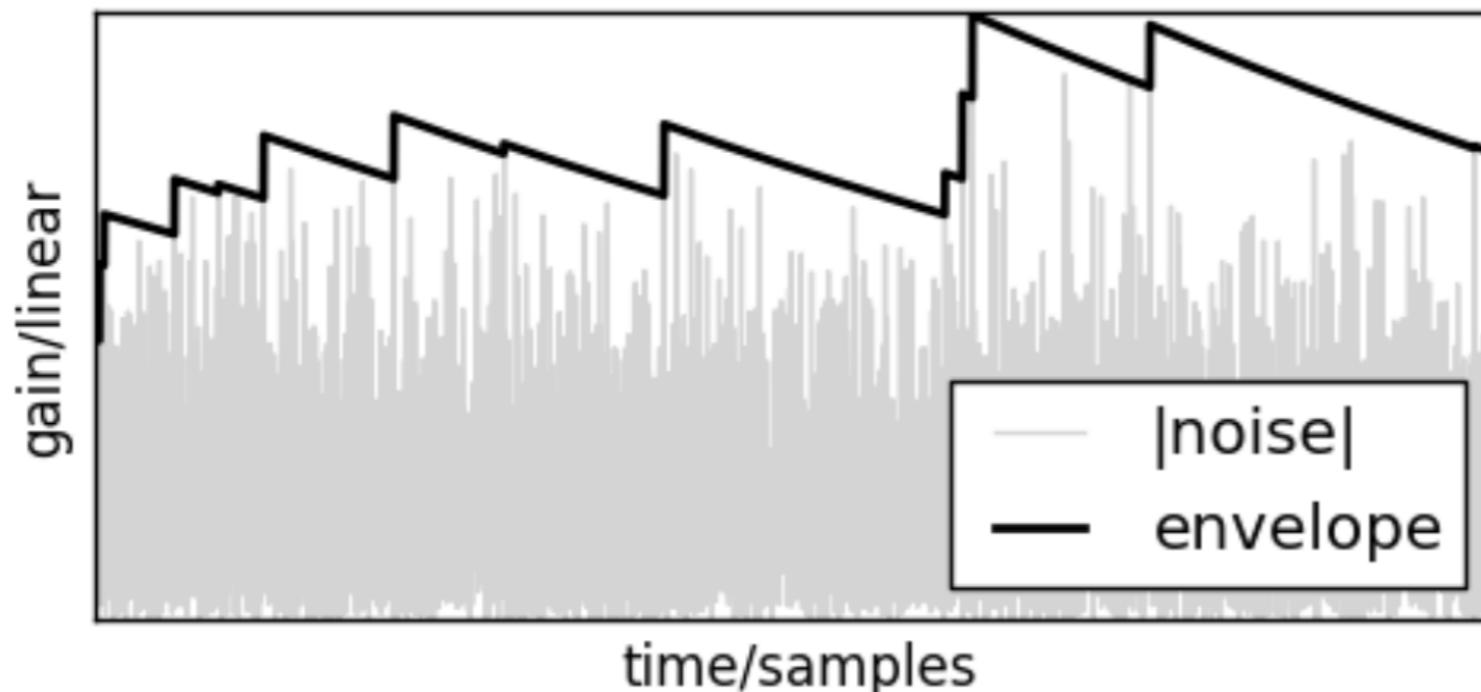
Projet 2 : Limiteur temps-réel



- **Comresseur** : effet audio visant à réduire la dynamique du signal (rapport entre plus fortes et plus faibles amplitudes).
- **Limiteur** : Compression qui vise à réduire le niveau des parties du signal qui dépassent durablement un seuil déterminé par l'utilisateur.
 - En diminuant l'amplitudes des parties les plus « fortes », c'est l'amplitude de l'ensemble du signal qui peut être réhaussée.
 - Utilisation pour le mastering, la diffusion radio —> effet « gros son »
 - Exemples sur <https://music.tutsplus.com/tutorials/a-beginners-introduction-to-limiters--audio-1071>

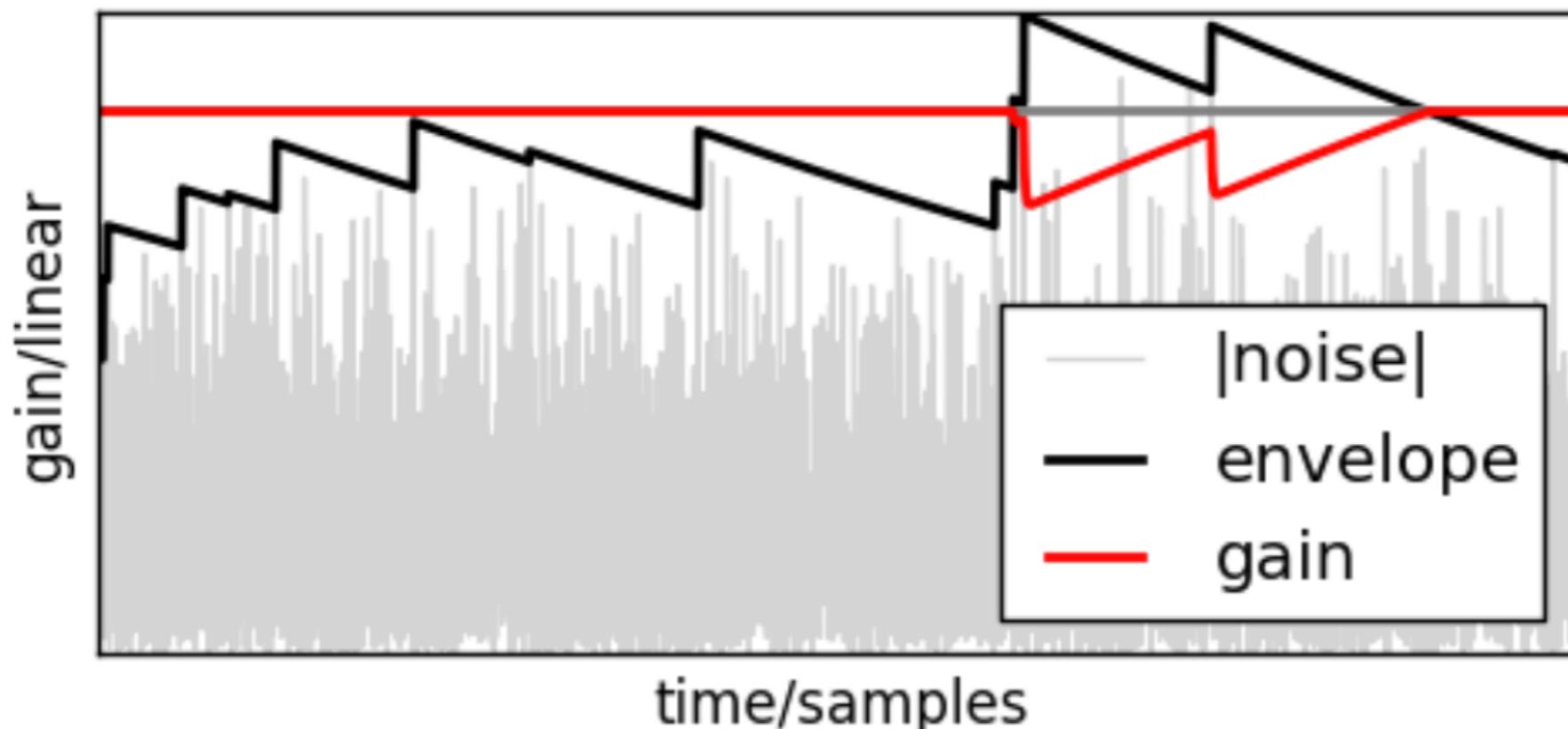
- Fonctionnement d'un limiteur :
 - Réduction dynamique et « smooth » du gain dès que l'amplitude dépasse un seuil fixé par l'utilisateur
 - Un simple seuillage des échantillons dont l'amplitude dépasse le seuil introduirait de fortes distorsions !
 - Retour « smooth » au signal original dès que l'amplitude de ce dernier re-devient inférieure au seuil
- Contrainte temps-réel : Estimation de l'amplitude moyenne « instantanée » calculée sur une portion de signal → latence (*lookahead*)
- Utilisation d'un **buffer circulaire**

- De nombreuses implémentations, nous allons en voir une « simple » (inspirée de *)
- Etape 1: on calcule une enveloppe e du signal signal s qui passe par tout les pics de s et dont l'amplitude décroît « doucement »
 - $e[n] = \max(|s[n]|, e[n - 1] \cdot f_r)$
 - f_r est le paramètre dit de « release » (avec $0 < f_r < 1$)



- Etape 2 : Calcul du gain instantané à appliquer à l'échantillon $s[n]$ pour diminuer son amplitude lorsque $e[n]$ dépasse un seuil K (fixé par l'utilisateur) (on suppose que $-1 < s[n] < 1$)

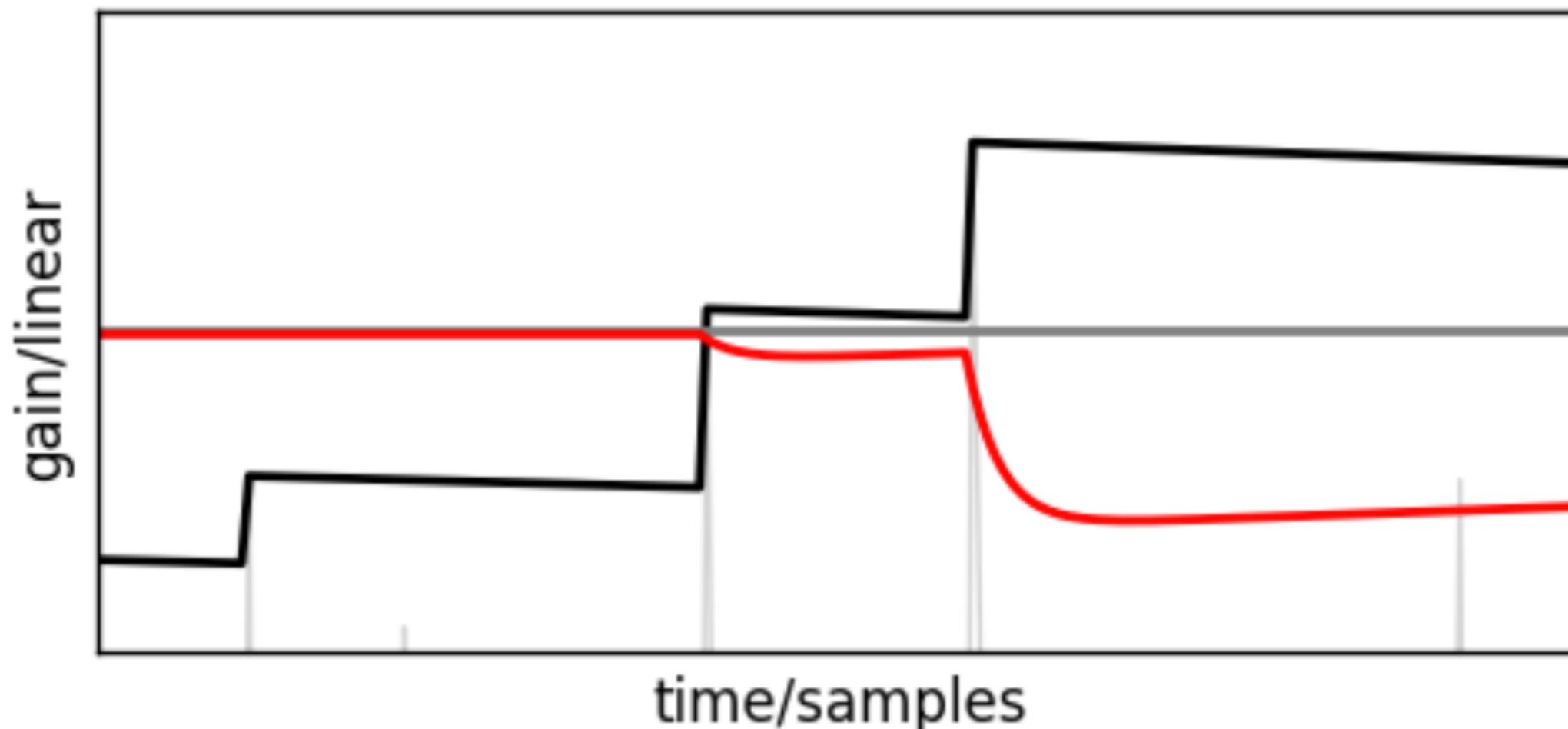
$$g_K[n] = \begin{cases} 1 & \text{pour } e[n] < K \\ 1 + K - e[n] & \text{pour } e[n] > K \end{cases}$$



- Lissage du gain instanée pour limiter les distorsions

$$g[n] = g[n - 1] \cdot f_a + g_K[n] \cdot (1 - f_a)$$

f_a est le paramètre d'attaque avec 0 < f_a << f_r



Déroulement des BE

- Travail en binôme
- Salle M373 (Minatec) (à vérifier)
- 16h: 4 Séances - Encadrement T. Hueber & L. Girin
- Si vous en avez, apporter vos casques ;-)
- Evaluation :
 - Démo en fin de la 4ème séance
 - 1 zip (nom1_nom2_tstr_2015.zip) contenant
 - Un README précisant la marche à suivre pour compiler et executer votre programme
 - un repertoire bin/ avec l'executable « reverb » et un executable « limiter »
 - un repertoire src/ contenant les sources (nettoyées et commentées) nécessaires à la compilation du programme
 - Rapport court (max 4 pages, nommé nom1_nom2_tstr_2015.pdf) détaillant vos choix d'implémentation et les résultats obtenus (fonctionnement général, temps de latence estimés pour chaque callback audio, etc).

- La pratique !

API RTAudio

- RtAudio is a set of C++ classes that provide a common API (Application Programming Interface) for realtime audio input/output across Linux, Macintosh OS-X and Windows operating systems. RtAudio significantly simplifies the process of interacting with computer audio hardware. It was designed with the following objectives:
 - object-oriented C++ design
 - simple, common API across all supported platforms
 - only one source and one header file for easy inclusion in programming projects
 - allow simultaneous multi-api support
 - support dynamic connection of devices
 - provide extensive audio device parameter control
 - allow audio device capability probing
 - automatic internal conversion for data format, channel number compensation, (de)interleaving, and byte-swapping
- RtAudio incorporates the concept of audio streams, which represent audio output (playback) and/or input (recording). Available audio devices and their capabilities can be enumerated and then specified when opening a stream. Where applicable, multiple API support can be compiled and a particular API specified when creating an RtAudio instance. See the API Notes section for information specific to each of the supported audio APIs.

```

#include "RtAudio.h"
(...)

int main()
{
    RtAudio adac;
    if ( adac.getDeviceCount() < 1 ) {
        std::cout << "\nNo audio devices found!\n";
        exit( 0 );
    }

    // Set the same number of channels for both input and output.
    unsigned int bufferBytes, bufferFrames = 512;
    RtAudio::StreamParameters iParams, oParams;
    iParams.deviceId = 0; // first available device
    iParams.nChannels = 2;
    oParams.deviceId = 0; // first available device
    oParams.nChannels = 2;

    try {
        adac.openStream( &oParams, &iParams, RTAUDIO_SINT32, 44100, &bufferFrames, &inout, (void *)
        * &bufferBytes );
    }
    catch ( RtAudioError& e ) {
        e.printMessage();
        exit( 0 );
    }

    bufferBytes = bufferFrames * 2 * 4;

    try {
        adac.startStream();

        char input;
        std::cout << "\nRunning ... press <enter> to quit.\n";
        std::cin.get(input);
    }
}

```

```

#include "RtAudio.h"
(...)

// Pass-through function.
int inout( void *outputBuffer, void *inputBuffer, unsigned int
nBufferFrames,
           double streamTime, RtAudioStreamStatus status, void
*data )
{
    // Since the number of input and output channels is equal, we can
do
    // a simple buffer copy operation here.
    if ( status ) std::cout << "Stream over/underflow detected." <<
std::endl;

    unsigned long *bytes = (unsigned long *) data;
    memcpy( outputBuffer, inputBuffer, *bytes );
    return 0;
}

```