Trace d'exécutions de systèmes temps-réel

Amine El Messaoudi - Encadrant : Julien Forget

IMA5 2018/2019 P10

Contents

Introduction 2
1- Contexte 3
    1-1- Presentation du projet 3
    1-2- Introduction au traçage 4
    1-3- Introduction à l'API ptask 5
2- Traçage 8
    2-1- Fonctionnement de LTTng 8
    2-2- Instrumentation et contrôle de LTTng 11
3- Travail d'instrumentation réalisé 16
    3-1- Besoin dans ptask 16
    3-2- Instrumentation de l'API 18
    3-3- Traçage du noyau 19
    3-4- Contrôle 20
4- Analyse du resultat 21
    4-1- Visualisation du CTF 21
    4-2- Parsing 22
    4-3- Exploitations des données 25
    4-4- Améliorations possibles 27
Conclusion 29
Sources 30

Introduction

Dans le cadre de ma formation, j'ai l'occasion de réaliser mon PFE en collaboration avec l'équipe de recherche **Émeraude** de **CRISTAL**, hébergé à l'**IRCICA**, encadré par **Julien Forget**.

L'Institut de Recherche sur les Composants logiciels et matériels pour l'Information et la Communication Avancée (**IRCICA**) est une Unité de Service et de Recherche (USR-3380) associant le CNRS et l'Université de Lille. Il comprend plusieurs équipes de recherche dont le groupe **Émeraude**.

Le groupe de recherche se propose d'étudier des architectures matérielles et leur utilisation efficace combinant des caractéristiques des circuits actuels et des caractéristiques, de nanocomposants émergents. L'objectif à long terme de cette équipe est d'obtenir un modèle de programmation nouveau pour des applications mobiles sur ces nouvelles machines, qui seront vraisemblablement massivement parallèles, hétérogènes, auront des contraintes de tolérance aux pannes et de consommation d'énergie à minimiser.

La notion de système temps-réel est fortement présente au sein du groupe de recherche. Mon projet au sein de L'IRCICA porte sur la réalisation d'un outil de trace d'exécutions de systèmes temps réel développé dans l'équipe. Ces systèmes temps-réel sont basés sur l'API ptask, une surcouche de pthread réalisé par **Giuseppe Lipari**, responsable de l'équipe. L'outil de trace utilisé pour ce projet est LTTng destiné aux systèmes Linux.

Ce document sert à présenter le projet, il contient une première partie sur l'introduction du projet et de son objectif. Puis un résumé de la phase recherche sur ptask et l'outil de trace LTTng, ainsi qu'une description de leurs fonctions déjà existantes. Le document présente dans une deuxième partie le travail réalisé pour implémenter l'outil de trace de ptask, puis le travail d'analyse post-traçage, allant du parsing à analyse jusqu'à l'exploitation et la visualisation des traces.

1- Contexte

1-1- Presentation du projet

1-1-1- Objectif

Un système temps réel est un système informatique chargé de contrôler un dispositif physique dans son environnement, ce à une vitesse adaptée à la vitesse d'évolution du dispositif. La spécificité principale d'un tel système est qu'il est aussi important de calculer le bon résultat que de le calculer au bon moment. Par exemple, dans le système de contrôle de vol d'un avion, il est essentiel que les calculs soient effectués suffisamment rapidement pour permettre une réaction à des perturbations extérieures (par exemple une bourrasque de vent) dans un laps de temps assurant la stabilité de l'avion. Les systèmes temps réel sont désormais présents dans de nombreux secteurs d'activités, tels que l'aéronautique, les centrales nucléaires, les chaînes de production d'usines ou encore la robotique mobile.

Un système temps réel est en général programmé sous la forme d'un ensemble de tâches exécutées de manière concurrente. Le comportement d'un tel système dépend non seulement du code de chaque tâche, mais aussi de leurs interactions et des dates auxquelles elles sont exécutées. Le débuggage d'un tel système nécessite de pouvoir étudier en détail ces interactions et ces dates d'exécution.

L'objectif de ce projet est de mettre au point un outil de traces d'exécution, et de statistiques basées sur ces traces, pour l'aide au débuggage d'un système temps réel. L'exécution du système temps réel en question est basée sur l'API ptask, une sur-couche temps réel aux pthreads POSIX. L'outil de trace s'appuiera sur le mécanisme de trace LTTng pour Linux. A l'aide du mécanisme de trace développé, nous souhaitons pouvoir obtenir des informations sur les préemptions subies par chaque tâche, le temps passé à exécuter chaque tâche, le nombre d'échéances manquées, ...

Le projet consiste en deux grandes étapes. Tout d'abord, étudier les mécanismes de trace LTTng existants pour le noyau Linux et l'espace utilisateur, ainsi que le code de l'API ptask, afin de déterminer le code d'instrumentation à ajouter à l'API ptask pour obtenir les traces voulues. Ensuite, développer un outil permettant d'exploiter les traces obtenues pour en extraire les informations décrivant le comportement temps réel (préemptions, temps d'exécution, ...).

1-1-2- Cahier de charges

- Se documenter sur l'outils de traces LTTng :
  - fonctionnement de l'outil.
  - instrumentation et contrôle.

- Se documenter sur l'API ptask.
  - fonctionnement de l'api.
  - degré d'exploitation de LTTng.
- Implémenter un outil d'extraction et d'analyse des traces.
  - instrumentation du code.
  - traçage et extraction des informations.
- Implémenter un outil de visualisation, et optimiser les résultat du traçage.
  - parsing des traces.
  - analyse des données.
  - exploitation des données.

1-2- Introduction au traçage

Le **traçage** est le processus de collecte d'informations sur l'activité dans un système en action. Cela se fait à l'aide d'outils spéciaux qui enregistrent les événements du système. Les programmes de traçage peuvent tracer simultanément les événements au niveau de l'application et de l'OS. Les informations qu'ils recueillent peuvent être utile pour diagnostiquer de multiples problèmes de système.

Le traçage est parfois comparé au *logging*. Il y a certainement des similitudes entre les deux, mais il y a aussi des différences. Avec le traçage, des informations sont écrites sur les événements de bas niveau. Ils se comptent par centaines, voire par milliers. Avec le *logging*, des informations sont écrites sur les événements de niveau supérieur, qui sont beaucoup moins fréquents.

Tout comme les *logs*, les données de traçage peuvent être lues telles quelles ; cependant, il est plus utile d'extraire des informations sur des applications spécifiques. Tous les programmes de traçage en sont capables. Parmi les mécanismes de trace du noyau linux, on trouve :

- **tracepoints** - un mécanisme qui fonctionne statiquement sur le code instrumenté.
- **kprobes** - un mécanisme de traçage dynamique utilisé pour interrompre le code d'un noyau à n'importe quel moment, appeler son propre gestionnaire, qui effectue les opérations nécessaires, puis reprend le code du noyau.

En excluant les solutions propriétaires, il existe quelques traceurs logiciels concurrents pour Linux :

- **dtrace4linux** est un portage de **DTrace** vers Linux. L'outil `dtrace` peut être utilisé pour obtenir une vue d'ensemble d'un système en cours

d'exécution, comme la quantité de mémoire, le temps CPU, le système de fichiers et les ressources réseau utilisées par les processus actifs.

- **perf** est un outil d'analyse des performances pour Linux qui extrait des informations à partir des compteurs de performances matérielles, des tracepoints, ainsi que d'autres compteurs et types de sondes.
- **strace** est un utilitaire en ligne de commande qui enregistre les appels système effectués par un processus utilisateur, ainsi que les signaux système et les changements d'état du processus.

La principale particularité de **LTTng** par rapport à ses concurrents est qu'il produit des traces corrélées du noyau et de l'espace utilisateur, tout en réduisant au minimum de coût. Il produit des fichiers de traces au format **CTF**, un format de fichier optimisé pour la production et l'analyse de données multi-gigaoctets. L'interface principale pour le contrôle de traçage est un outil en ligne de commande unique nommé `lttng`.

Ce dernier peut créer plusieurs sessions de traçage, activer et désactiver des critères de traçage à la volée, filtrer efficacement des critères de traçage avec des expressions utilisateur personnalisées, démarrer et arrêter le traçage, et bien plus encore. **LTTng** peut enregistrer les traces sur le système de fichiers ou les envoyer sur le réseau, et les conserver totalement ou partiellement.

1-3- Introduction à l'API ptask

Ptask est une bibliothèque C pour le développement rapide de tâches périodiques et apériodiques en temps réel sous Linux. L'API est une surcouche de la bibliothèque Pthread dans le but de simplifier la création de threads avec des paramètres de timing typiques, comme les périodes et les délais. Les fonctions Ptask permettent, entre autres, de rapidement :

- créer des tâches périodiques et apériodiques ;
- préciser les contraintes temporelles telles que les périodes et les délais relatifs ;
- surveiller les échéances manquées ;
- surveiller les temps d'exécution moyens et les temps d'exécution dans le pire des cas ;
- activer des tâches avec des décalages spécifiques ;
- gérer les groupes des tâches ;
- gérer les changements de mode.

Dans la bibliothèque sont implémentés de nouveaux types, notamment pour des variables temporelles, et des structures pour stocker les paramètres d'une tâche. Et des fonctions permettant la manipulation des tâches.

Ci-dessous un exemple d'utilisation de ptask ou trois tâches sont créées :

```
#include "ptask.h"
#include "calibrate.h"
#include <pthread.h>
#include <sched.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define PER 50  //task period
#define RNT 25  //task runtime
#define DREL 25 //task deadline
#define PRIO 30 //task priority
#define TT 1    //task forced time

void init() {
   srand(time(NULL));
   ptask_init(SCHED_DEADLINE, GLOBAL, PRIO_INHERITANCE);
}

void periodic_task() {
   for (int i=0; i<1000; i++) {
       work_for(TT, MILLI); // do useful things as a task for TTms
       ptask_wait_for_period(); // wait for the next period
   }
}

void aperiodic_task() {
   while (1) {
       work_for(TT, MILLI); // do useful things as a task for TTms
       ptask_wait_for_activation(); // wait for activation
   }
}

void activ_aper_task()
{
   int task_to_activate = *((int*) ptask_get_argument());
   int i = 0;
   while(1) {
       if (i%3 == 0) work_for(15, MILLI);
       else work_for(5, MILLI);

       if (i%2 == 0) ptask_activate(task_to_activate);

       i++;
```

```
        ptask_wait_for_period();
    }
}


int main(void) {
    int i;
    int aper, act_aper;

    init();

    tpars params = TASK_SPEC_DFL;
    params.period = tspec_from(PER, MILLI);
    params.rdline = tspec_from(DREL, MILLI);
    params.measure_flag = 1;
    params.processor = 0;
    params.act_flag = NOW;
    params.runtime = tspec_from(RNT, MILLI);
    params.priority = PRIO;

    i = ptask_create_param(periodic_task, &params);
    if (i != -1) printf("Task %d created and activated\n", i);
    else exit(-1);

    params.priority = PRIO - i;
    params.act_flag = DEFERRED;

    aper = ptask_create_param(aperiodic_task, &params);
    if (aper != -1) printf("Task %d created and activated\n", aper);
    else exit(-1);

    params.arg = &aper;
    act_aper = ptask_create_param(activ_aper_task, &params);
    if (act_aper != -1) printf("Task %d created and activated\n", act_aper);
    else exit(-1);
    ptask_activate_at(act_aper, 100, MILLI);

    while(1) ;

    return 0;
}
```

Dans cet exemple d'utilisation, trois tâches sont créées :

- une première tâche périodique periodic_task() lancée au moment de la création, s'exécute chaque 50ms pendant 5ms;

- une deuxième tâche apériodique aperiodic*task() créée, mais qui nécessite un 'ptask*activate()' pour être exécuté pendant 5ms;
- une dernière tâche périodique activ*aper*task() qui active la tâche apériodique une fois sur deux et qui dure 15ms une fois sur trois.

Pour plus de détails sur ptask, se référer au répertoire git.

2- Traçage

2-1- Fonctionnement de LTTng

2-1-1- Introduction

Le **traceur** kernel **"Linux Trace Toolkit : new generation"** est un logiciel "*toolkit*" *open source* qui peut être utilisé pour tracer simultanément le noyau Linux, les applications utilisateur et les bibliothèques utilisateur. **LTTng** se compose entre autres de :

- Modules du noyau pour tracer le noyau Linux.
- Bibliothèques partagées pour tracer les applications utilisateur écrites en C ou C++.
- Un module noyau pour tracer les scripts shell et autres applications utilisateur sans mécanisme d'instrumentation dédié.
- Daemons et un outil en ligne de commande, `lttng`, pour contrôler les traceurs LTTng.

Entre autre, il peut effectuer les tâches suivantes :

- Analyser les interactions interprocessus dans le système.
- Analyser les interactions application-kernel dans l'espace utilisateur.
- Mesurer le temps que le noyau passe à répondre aux demandes d'application.
- Analyser le fonctionnement du système sous des charges de travail élevées.

2

-1-2- Installation

LTTng est un ensemble de composants logiciels qui interagissent pour instrumenter le noyau Linux et les applications utilisateur, et pour contrôler le traçage (démarrer et arrêter le traçage, activer et désactiver les règles d'événements, et le reste). Ces composants sont regroupés dans les packages suivants :

- **LTTng-tool** : Bibliothèques et interface en ligne de commande pour contrôler le traçage.
- **LTTng-modules** : Modules du noyau Linux pour instrumenter et tracer le noyau.
- **LTTng-UST** : Bibliothèques et paquets Java/Python pour instrumenter et tracer les applications utilisateur.

Pour installer LTTng, se référer à la documentation LTTng.

2-1-3- Concepts de bases

Du point de vue de l'utilisateur, le système LTTng est construit sur quelques concepts, sur lesquels `lttng` *command-line tool* fonctionne en envoyant des commandes à la session *deamon*. Comprendre comment ces objets sont liés les uns aux autres est essentiel pour maîtriser l'outil. Les concepts de base sont :

Session de trace

Une session de traçage est un dialogue dynamique entre l'utilisateur et la session **deamon**. La commande `lttng create` permet de créer une session de traçage, et chaque session a :

- son propre nom.
- ses propres fichiers de trace.
- son propre état d'activité (démarré ou arrêté).
- son propre mode (local, streaming réseau, snapshot ou en direct).
- ses propres *channels* qui ont leurs propres règles d'événements.

Une session de traçage dure de la commande `lttng create` à la commande `lttng destroy`.

Domaine de trace

Un domaine de traçage est un espace pour les sources d'événements. Un domaine de traçage a ses propres propriétés et caractéristiques. Il y a actuellement cinq domaines de traçage disponibles :

- noyau Linux
- espace utilisateur
- `java.util.logging` (JUL)
- log4j
- Python

On peut spécifier un domaine de traçage lors de l'utilisation de certaines commandes pour éviter toute ambiguïté. Par exemple, puisque tous les domaines supportent les *tracepoints* nommés comme sources d'événements.

Il est également possible de créer des canaux dans les domaines de traçage du noyau Linux et de l'espace utilisateur. Les autres domaines de traçage ont un seul canal par défaut.

Dans notre cas, seuls les domaines noyaux linux et l'espace utilisateur sont tracés.

Channel* et *ring buffer

Un *channel* est un objet responsable d'un ensemble de *buffers* circulaires. Chaque *buffer* circulaire est divisé en plusieurs sous-*buffers*. Lorsqu'un traceur LTTng émet un événement, il peut l'enregistrer dans un ou plusieurs sous-*buffers*. Les attributs d'un *channel* déterminent ce qu'il faut faire lorsqu'il n'y a plus de place pour un nouvel enregistrement d'événement parce que tous les sous-*buffers* sont pleins.

Un *channel* est toujours associé à un domaine de traçage. Et possède également des règles d'événement.

Lorsqu'un événement se produit, LTTng l'enregistre dans un sous-*buffer* spécifique du *buffer* circulaire d'un *channel* spécifique. Quand il n'y a plus de place dans un sous-*buffer*, le traceur le marque comme consommable et un autre sous-*buffer* vide commence à recevoir les enregistrements d'événements suivants. Un *deamon* consommateur finit par consommer le sous-*buffer* marqué. Par défaut, les modules LTTng et LTTng-UST sont des traceurs non bloquants : lorsqu'aucun sous-*buffer* vide n'est disponible, il est acceptable de perdre des enregistrements d'événements alors que l'alternative serait de causer des retards importants dans l'exécution de l'application instrumentée. Mais à partir de LTTng 2.10, le traceur d'espace utilisateur LTTng, LTTng-UST, supporte un mode de blocage.

Points d'instrumentation, règles d'*event*, *event*, enregistrement d'*event*

Une règle d'événement est un ensemble de conditions qui doivent toutes être remplies pour que LTTng puisse enregistrer un événement.

Les conditions sont créées lors de la création de la règle d'événement.

Et une règle d'événement doit toujours être attachée au canal lorsque qu'il est créé.

Lorsqu'un événement passe les conditions d'une règle d'événement, LTTng l'enregistre dans l'un des sous-tampons du canal attaché.

Les conditions disponibles, à partir de LTTng 2.10, sont :

- La règle d'événement est activée.
- Le type du point d'instrumentation.

- Le nom du point d'instrumentation.
- Le niveau du log du point d'instrumentation.
- Les champs du *playload* de l'événement satisfont une expression de filtre.

Dans notre cas, trois événements -dont deux tracepoints- sont enregistrées. Ces événements correspondent au besoin de traçage et sont expliqués en détail par la suite.

2-2- Instrumentation et contrôle de LTTng

2-2-1- Composants

L'écosystème LTTng est composé de plusieurs composantes, qui interagissent avec les applications utilisateur, avec le kernel Linux, puis avec l'utilisateur. Le projet LTTng intègre LTTng-tools, LTTng-UST, LTTng-modules. La figure ci-dessous résume la plupart des fonctionnalités :

Interface en ligne de commande pour le contrôle du traçage :

Le traçage commence par une commande de l'utilisateur -pointillés rouge- à travers l'outil en ligne de commande `lttng` est l'interface utilisateur standard pour contrôler les sessions de traçage LTTng. Il fait partie des **LTTng-tools**, et est lié à `liblttng-ctl` pour communiquer avec un ou plusieurs **session-deamons** dans les coulisses.

```
lttng <GENERAL OPTIONS> <COMMAND> <COMMAND OPTIONS>
```

Bibliothèque pour le contrôle du traçage :

La bibliothèque de contrôle LTTng, `liblttng-ctl`, est ensuite utilisée pour communiquer avec le *session-deamon* en utilisant une **API C** qui cache les détails du protocole sous-jacent. La bibliothèque fait partie de **LTTng-tools**. Pour l'utiliser en C ou C++, il suffit d'inclure son en-tête "master" :

```
#include <lttng/lttng.h>
```

Bibliothèque de traçage pour l'espace utilisateur :

Lors d'un traçage dans le domaine espace utilisateur la bibliothèque de traçage de l'espace utilisateur, `liblttng-ust`, est utilisé. Il reçoit des commandes d'un *session-deamon*, par exemple pour activer et désactiver des points d'instrumentation spécifiques, et écrit des enregistrements d'événements dans des *buffers* circulaires partagés avec un *consumer-deamon*. `liblttng-ust` fait partie de **LTTng-UST**.

Modules LTTng pour le kernel :

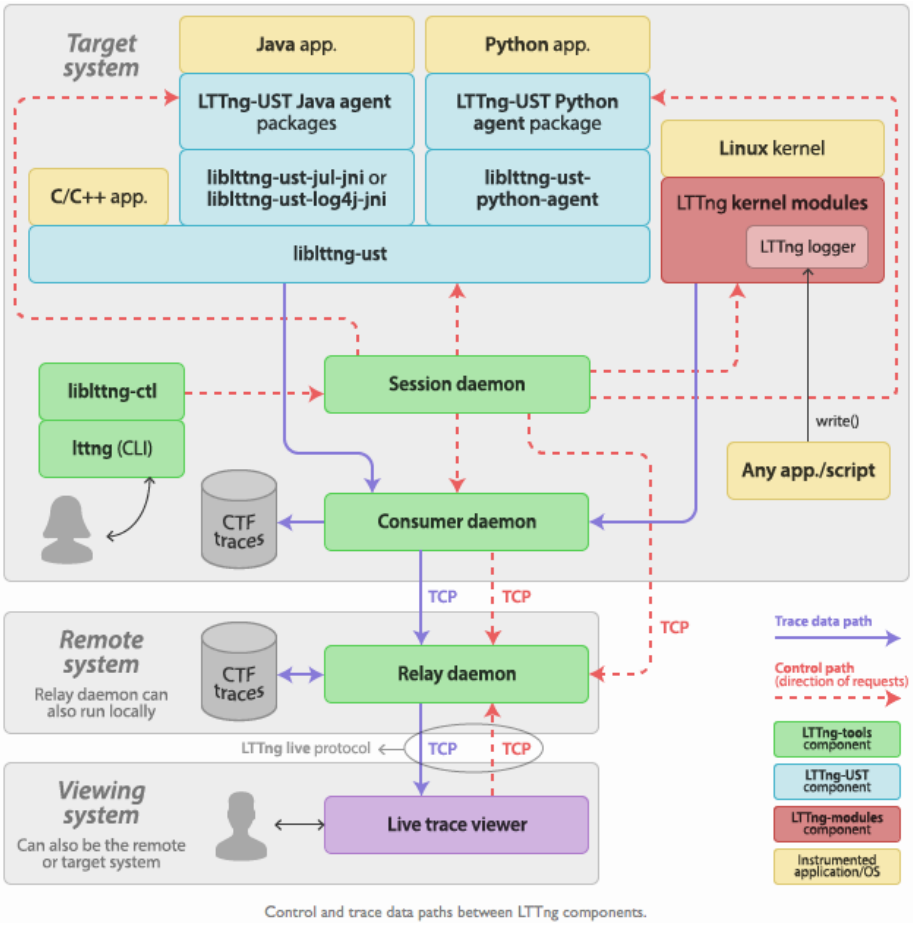


Figure 1: Composants de LTTng

Les modules du noyau LTTng sont un ensemble de modules du noyau Linux qui implémentent le traceur du noyau du projet LTTng. Les modules du noyau LTTng font partie de **LTTng-modules**.

Session deamon :

Le *session-deamon* est un démon responsable de la gestion des sessions de traçage et du contrôle des différents composants de LTTng. Le démon de session fait partie de **LTTng-tools**. Il envoie des demandes de contrôle à (et reçoit des réponses de contrôle de) :

- La bibliothèque de traçage de l'espace utilisateur.
- Les agents de traçage de l'espace utilisateur.
- Le traceur du noyau.
- Le démon de "consommation".
- Le démon relai.

***Consumer deamon* :**

Le démon consommateur, `lttng-consumerd`, est un démon qui partage des *buffers* avec les applications utilisateur ou avec les modules du noyau LTTng pour collecter les données de trace et les envoyer à un emplacement (sur disque ou à un démon relais sur le réseau). Le démon consommateur fait partie de **LTTng-tools**.

2-2-2- Instrumentation

Plusieurs méthodes permettent d'instrumenter un logiciel de traçage LTTng. Le plus simple est de placer manuellement les points d'instrumentation, appelés *tracepoints*, dans le code source du logiciel. Il est également possible d'ajouter dynamiquement des points d'instrumentation dans le domaine du traçage du noyau Linux. Pour un système temps-réel, il faudra également tracer le noyau Linux, et notamment les appels système qui gèrent les Threads. Dans ce cas, les besoins en instrumentation sont déjà couverts en majorité par les points de trace intégrés du noyau Linux de LTTng. Pour instrumenter une application utilisateur, il faudra utiliser la bibliothèque `lttng-ust`, en suivant les étapes :

Créer les fichiers sources d'un package fournisseur tracepoint

Un fournisseur de points de trace (ou **tracepoints provider**) est un ensemble de fonctions compilées qui fournissent des points de trace à une application, le type de point d'instrumentation supporté par LTTng-UST. Ces fonctions peuvent émettre des événements avec des champs définis par l'utilisateur et sérialiser ces événements en tant qu'enregistrements d'événements dans un ou plusieurs sous-tampons de canaux LTTng-UST. La macro `tracepoint()`, que vous insérez dans le code source d'une application utilisateur, appelle ces fonctions.

Dans notre cas, un *tracepoints provider* qui correspond aux événements et informations qu'il faut extraire est créé. (Exemple dans la section suivante.)

Un package fournisseur tracepoint est un fichier objet (`.o`) ou une bibliothèque partagée (`.so`) qui contient un ou plusieurs fournisseurs tracepoint. Ses fichiers sources sont :

- Un ou plusieurs en-tête(s) de fournisseur tracepoint (`.h`).
- Une source d'emballage fournisseur de tracepoint (`.c`).

Un package fournisseur tracepoint est lié dynamiquement avec `liblttng-ust`, le traceur d'espace utilisateur LTTng, au moment de l'exécution.

Construire et lier un package fournisseur de tracepoint et une application

Une fois un ou plusieurs fichiers d'en-tête de fournisseur de tracepoint et un fichier source de paquet de fournisseur de tracepoint prêts, il faut créer le paquet de fournisseur de tracepoint en compilant son fichier source. A partir de là, plusieurs choix de construction et d'exécution sont possibles.

Dans le cas où l'application instrumentée est liée statiquement à l'objet package du fournisseur tracepoint :

1. Compiler le fichier source du paquet fournisseur tracepoint : `gcc -I. -c hello-tp.c`
2. Dans `hello.c`, avant d'inclure `hello-tp.h`, ajouter : `#define TRACEPOINT_DEFINE`
3. Compiler le fichier source de l'application : `gcc -c hello.c`
4. Créer l'application : `gcc -o hello hello.o hello-tp.o -llttng-ust -ldl`
5. Lancer l'application : `./hello`

L'application instrumentée peut aussi être liée statiquement à une archive du package du fournisseur tracepoint (`.a`), un objet partagé du package du fournisseur tracepoint, ...

Pour plus de détails, se référer à la documentation.

2-2-3- Contrôle

Créer une session démon

Chaque utilisateur Unix doit avoir son propre démon de session en cours d'exécution pour tracer les applications utilisateur. Le démon de session que l'utilisateur root lance est le seul autorisé à contrôler le traceur du noyau LTTng.

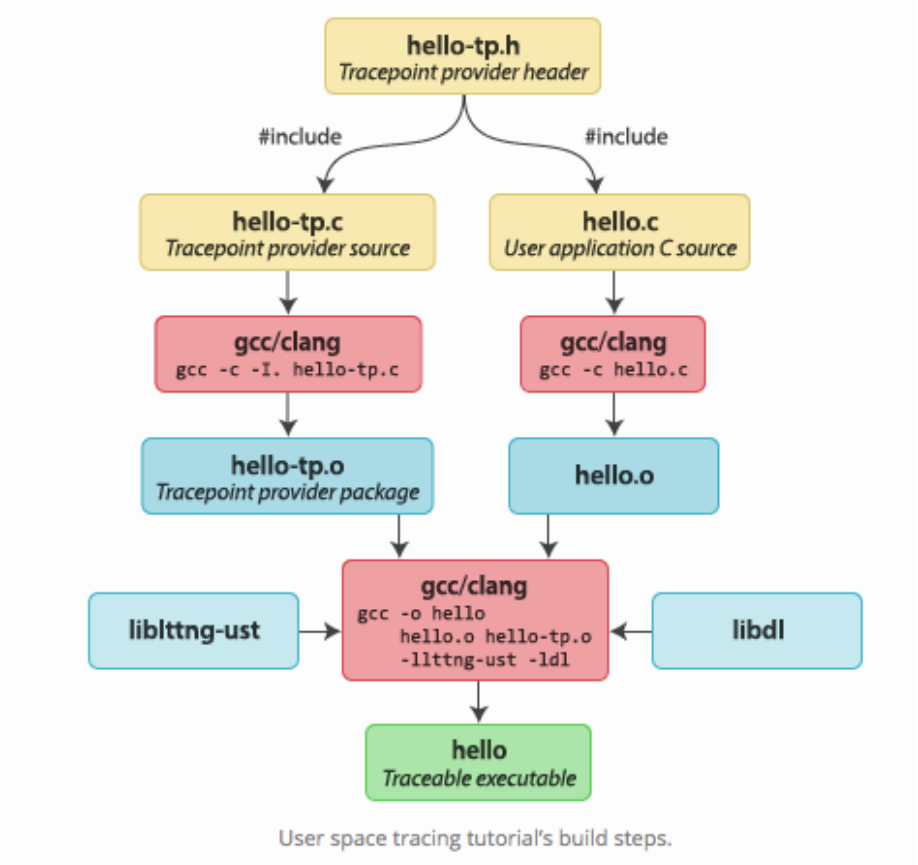


Figure 2: Étape de construction d'une aplication

Les utilisateurs qui font partie du groupe de traçage peuvent contrôler le démon de session racine.

```
lttng-sessiond --daemonize
```

Pour arrêter un démon de session, il faut utiliser `kill(1)` sur son ID de processus (signal `TERM` standard).

Créer et détruire une session de trace

Presque toutes les opérations de contrôle LTTng se déroulent dans le cadre d'une session de traçage, qui est le dialogue entre le démon de session et l'utilisateur.

```
lttng create my-session --output=/tmp/some-directory
```

LTTng ajoute la date de création au nom de la session de traçage créée. Il écrit les traces d'une session de traçage dans `$LTTNG_HOME/lttng-trace/name` par défaut, où name est le nom de la session de traçage (avec la variable d'environnement LTTNG_HOME par défaut est `$HOME` si elle n'est pas définie). Il existe également des commandes lttng qui opèrent sur la session courante, et qui permettent de démarrer, arrêter, détruire, ajouter des chaines ou *events*...

Liste des points d'instrumentation et statut

Le démon de session peut interroger les applications utilisateur en cours d'exécution et le noyau Linux pour obtenir une liste des points d'instrumentation disponibles. Pour le domaine de traçage du noyau Linux, ce sont des tracepoints et des appels système.

```
lttng list --userspace
lttng list --kernel --syscall
```

Pour obtenir l'état de la session de traçage en cours, c'est-à-dire ses paramètres, ses canaux, ses règles d'événement et leurs attributs :

```
lttng list my-session
```

3- Travail d'instrumentation réalisé

3-1- Besoin dans ptask

Les critères d'un système **temps réel** peuvent être classés en trois catégories : dur, ferme ou mou. Il existe de multiples modèles mathématiques pour représenter un système d'ordonnancement, la plupart des implémentations d'algorithmes d'ordonnancement en temps réel sont modélisées pour l'implémentation de configurations uniprocesseurs ou multiprocesseurs. Les algorithmes utilisés dans l'analyse de planification "peuvent être classés comme préventifs ou non préventifs".

Dans la figure ci-dessous chaque processus correspond à une tâche périodique avec une période de 1ms. Ces process s'exécutent avec préemptions, ce qui veut dire que la tâche jaune peut interrompre l'orange dans la troisième période si celle-ci est plus prioritaire.

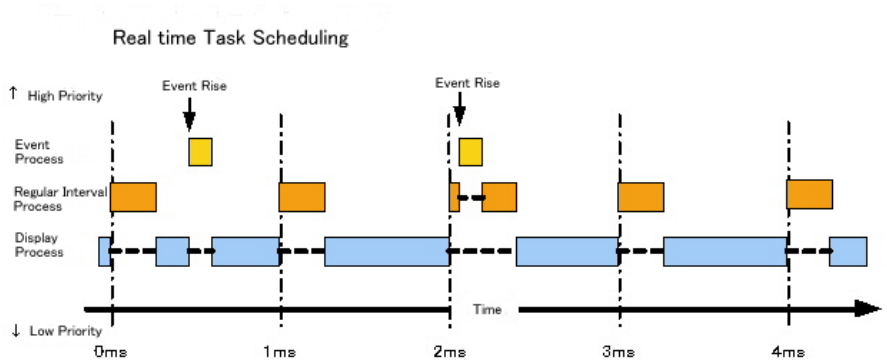


Figure 3: Ordonnancement tâches temps-réel

Les informations qu'il faudra extraire du traçage devront contenir pour chaque tâche :

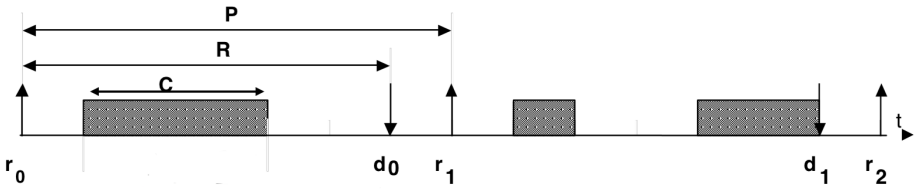


Figure 4: Paramètres d'ordonnancement

- L'indice de la tâche : identifiant qui nous permet de reconnaître la tâche.
- La date de réveil r0 et l'échéance d0 (intervalle R dans lequel la tâche doit être exécuter).
- La date de début et de fin de C : début d'exécution de chaque tâche.
- Les dates de préemption.

Les taches de l'API ptask représentent un thread chacun, lancé à la creation de la tâche. Si la tâche est lancée avec le flag **NOW**, elle est immédiatement activée ; et si elle l'est avec **DEFERRED**, la tâche sera bloquée sur `wait_for_activation()` jusqu'à ce qu'une `ptask_activate()` soit appelée par une autre tâche.

Et la fonction `ptask_wait_for_period()` permet d'attendre la prochaine période après la fin d'une exécution.

En plus de ces informations, il faut savoir le `pid` de chaque tâche et l'id du cpu qui exécute la tâche, ces informations permettront la reproduction du schéma d'ordonnancement par cpu.

Pour avoir des informations sur la tâche, et sur le début et la fin de chaque période, il faut commencer par implémenter des tracepoints dans l'API, notamment dans la fonction `ptask_wait_for_period()` pour une tâche périodique.

3-2- Instrumentation de l'API

Les fonctions de ptask implémentées sont les fonctions `wait_for_activation()`et `wait_for_period()`. Le fait d'insérer des tracepoints dans ces fonctions va permettre de déterminer la fin de chaque tâche, et le début de chaque période.

```
tracepoint(ptask_provider, ptask_tracepoint, pid, tid, ptask_idx,
    flag, state, tspec_to_rel(&now, MICRO), _tp[ptask_idx].priority,
    tspec_to(&_tp[ptask_idx].period, MICRO), tspec_to(&_tp[ptask_idx]
    .deadline, MICRO));
```

Ce tracepoint implémenté permet de récupérer :

- Le pid de la tâche,
- Le tid de la tâche,
- L'index de la tâche,
- Le flag de création de la tâche,
- La période,
- La priorité,
- La deadline.

Il est compris par LTTng grâce à un **tracepoints provider**, le fichier source `.h` de ce dernier est sous la forme :

```
#undef TRACEPOINT_PROVIDER
#define TRACEPOINT_PROVIDER ptask_provider

#undef TRACEPOINT_INCLUDE
#define TRACEPOINT_INCLUDE "./tp.h"

#if !defined(_TP_H) || defined(TRACEPOINT_HEADER_MULTI_READ)
#define _TP_H

#include <lttng/tracepoint.h>

TRACEPOINT_EVENT(
    ptask_provider,
```

```
    ptask_tracepoint,
    TP_ARGS(
        int, pid,
        int, tid,
        int, i,
        char*, flag,
        char*, state,
        long, time,
        int, priority,
        int, period,
        int, deadline
    ),
    TP_FIELDS(
        ctf_integer(int, ptask_pid, pid)
        ctf_integer(int, ptask_tid, tid)
        ctf_integer(int, ptask_index, i)
        ctf_string(ptask_flag, flag)
        ctf_string(ptask_state, state)
        ctf_integer(long, ptask_time, time)
        ctf_integer(int, ptask_priority, priority)
        ctf_integer(int, ptask_period, period)
        ctf_integer(int, ptask_deadline, deadline)
    )
)

TRACEPOINT_LOGLEVEL(ptask_provider, ptask_tracepoint, TRACE_INFO)

#endif /* _TP_H */
#include <lttng/tracepoint-event.h>
```

Le tracepoint a était implémenté dans un fork de l'api ptask, les fichiers du tracepoint provider ont été intégrés au projet. L'api utilise l'utilitaire cmake pour le contrôle du processus de compilation du code, ce qui veut dire que les fichiers de configuration cmake doivent également être modifiés de façon à intégrer la bibliothèque liblttng-ust, et à compiler le tracepoint provider. Ces changements ont été apportés à l'api via un `pull request`.

En plus des informations sur la tâche, le tracepoint ajouté permet de déterminer la date de fin et de début de chaque tâche à partir du deuxième réveil. Pour pouvoir déterminer le premier réveil de la tâche il faudra repérer la création du thread, et donc effectuer un traçage dans le noyau en plus de l'espace utilisateur.

3-3- Traçage du noyau

Une tâche correspond à ce qu'on appelle **processus légers**, qui sont une généralisation des concepts de **processus** et de **threads**, et peuvent être utilisés pour

les mettre en œuvre. Linux a un appel système `clone` qui est utilisé pour créer de nouveaux processus légers. Lors d'un `clone`, on peut fournir divers *flags* avec, qui indiquent ce qui sera partagé entre le nouveau processus et le processus existant. Les *threads* utilisent le `clone` pour créer des processus qui partagent leur espace d'adressage, des fichiers ouverts, des gestionnaires de signaux...

Le traçage de l'appel système `clone` et son analyse peuvent indiquer la création de la tâche et donc la première date de réveil.

L'appel système `clone` dans LTTng renvoie deux traces : `syscall_entry_clone` et `syscall_exit_clone`. La première contient le *flag*, un id parent, et un id enfant ; et la deuxième contient la valeur de retour du clone.

Une fois que nous avons les dates de début, et de fin de chaque tâche, il reste uniquement à pouvoir déterminer les préemptions entre les tâches.

Grâce à la bibliothèque `LTTng-modules`, le noyau est déjà instrumenter avec plusieurs tracepoints. Pour pouvoir détecter les changements de processus, et donc la préemption entre les tâches, il suffit d'activer le tracepoint qui correspond : `sched_switch`. Le traçage et l'analyse de ce dernier permettent de récupérer la date de bascule entre deux tâches.

3-4- Contrôle

Après avoir lancé une session démon et créé une session de traçage, il suffit simplement d'activer les *events* :

- Appel système `clone` dans le noyau.
- Tracepoint du noyau `sched_switch`.
- Tracepoint de l'espace utilisateur créé.

Il faut également ajouter un context dans le noyau permettant de récupérer le `tid` et `pid` lors du traçage. Pour l'espace utilisateur, ces informations ont été ajoutées manuellement lors de l'implémentation.

```
lttng create <NAME>
lttng enable-event --kernel --syscall clone
lttng enable-event --kernel sched_switch
lttng enable-event --userspace ptask_provider:ptask_tracepoint
lttng add-context --kernel --type=pid --type=tid
lttng start
```

Les traces sont enregistrées dans deux dossiers séparés : *kernel* et espace utilisateur, avec chacun un seul *channel*. Le *channel* est créé lorsque la session est créée, et les paramètres du *channel*, des *buffers*, et également les paramètres d'enregistrement d'événements utilisés sont ceux par défaut.

Il est possible de revoir les paramètres de création des canaux de traçage pour qu'ils soient le moins coûteux pour l'application temps réel.

Après, il faut stopper la session et la détruire :

```
lttng stop
lttng destroy
```

Les traces sont enregistrées au format CTF dans le répertoire par défaut, sous la forme :

```
[12:20:16.002620588] (+0.000004208) debian-vm syscall_exit_clone:
    { cpu_id = 0 }, { pid = 16226, tid = 16308 }, { ret = 0 }
[12:20:16.002593777] (+0.000034078) debian-vm syscall_entry_clone:
    { cpu_id = 0 }, { pid = 16226, tid = 16226 }, { clone_flags =
    0x3D0F00, newsp = 0x7FC06FF5CF70, parent_tid = 0x7FC06FF5D9D0,
    child_tid = 0x7FC06FF5D9D0 }
[12:20:16.842354195] (+0.000009715) debian-vm ptask_provider:
    ptask_tracepoint: { cpu_id = 0 }, { ptask_flag = "DEFERRED",
    ptask_state = "b_wait_period", ptask_pid = 16226, ptask_tid =
    16308, ptask_index = 0, ptask_time = 150276421, ptask_priority
    = 80, ptask_period = 20000, ptask_deadline = 20000 }
[12:20:16.842358357] (+0.000004162) debian-vm sched_switch: { cpu_id
    = 0 }, { pid = 16226, tid = 16308 }, { prev_comm = "ball",
    prev_tid = 16308, prev_prio = -81, prev_state = 1, next_comm =
    "ball", next_tid = 16309, next_prio = 20 }
```

L'exemple ci-dessus présente les quatre types d'événements nécessaires à la représentation de l'ordonnancement :

- syscall*exit*clone.
- syscall*entry*clone.
- ptask_tracepoint.
- sched_switch.

4- Analyse du resultat

4-1- Visualisation du CTF

Le **Common Trace Format** (CTF) est un format de trace binaire conçu pour être très rapide à écrire sans compromettre une grande flexibilité. Il permet de générer nativement des traces à partir de n'importe quelle application ou système C/C++, ainsi qu'à partir de composants *bare-metal* (matériels).

Avec CTF, tous les en-têtes, contextes et champs d'événements écrits dans des fichiers binaires sont décrits à l'aide d'un langage déclaratif de type C personnalisé

appelé Trace Stream Description Language (TSDL). TSDL permet de décrire de nombreuses mises en page de flux de traces binaires grâce à la large gamme de types de champs disponibles au CTF.

L'avantage de ce format, surtout pour un système temps-réel, est la rapidité d'écriture. C'est un format optimisé pour l'analyse de plusieurs gigabyte de données.

L'outil permettant de convertir le CTF en texte est Babeltrace. Il s'agit d'une application de conversion de trace / bibliothèque qui est capable de lire et d'écrire CTF, en supportant presque toutes ses fonctionnalités spécifiées.

Le résultat du traçage figurant dans le chapitre précédent est généré à partir de `babeltrace`, avec une commande qui ressemble à :

```
babeltrace /root/lttng-traces/[app_name]/
```

Il existe également un autre outil : **Trace Compass** permettant de visualiser et d'analyser tout type de logs ou de traces. L'avantage de cet outil java est de fournir des vues, des graphiques, des métriques, etc. pour aider à extraire des informations utiles des traces, d'une manière plus conviviale et informative que les grandes décharges de texte.

Cet outil a été utilisé lors des tests, et a permis une meilleure compréhension de l'ordonnancement des tâches de ptask. L'exemple qui suit illustre le traçage d'un programme `test` qui crée un simple thread de tid 22744.

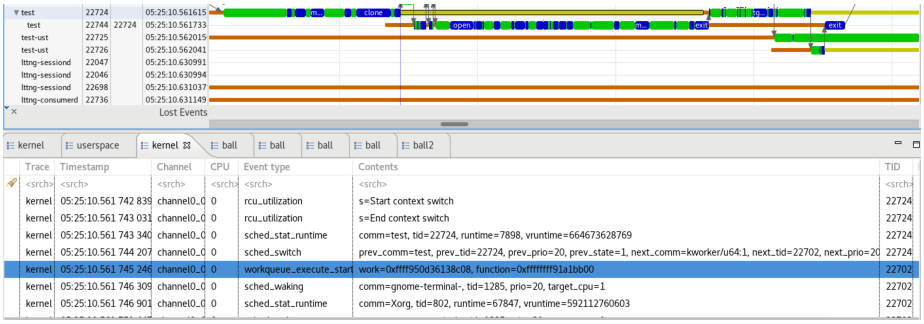


Figure 5: Test traçage thread

4-2- Parsing

4-2-1- Parsing préliminaire

Après la production de la trace sous forme de texte, nous obtenons un résultat peu lisible. Le fichier de trace, et plus précisément du domaine kernel, contient des événements `clone` et `sched_switch` qui non seulement correspondent à

l'application tracée, mais également à tous les processus qui tournent sur la machine.

Lttng permet d'indiquer les pids des processus à tracer via la commande :

```
lttng track --kernel --pid=`ps axf |grep <app> |grep -v grep |grep
    -v sudo |cut -d" " -f1`
```

Cette commande peut permettre d'écarter tout les processus parasites qui s'exécutent en même temps sur la machine. Elle doit être lancée juste avant le `lttng start`, et nous permet d'avoir un résultat similaire à :

```
...
[12:20:15.327313668] (+0.000833000) debian-gnu-linux-vm sched_switch:
    { cpu_id = 0 }, { pid = 16226, tid = 16226 }, { prev_comm = "ball"
    , prev_tid = 16226, prev_prio = 20, prev_state = 0, next_comm =
    "java", next_tid = 1607, next_prio = 20 }
[12:20:15.327532761] (+0.000219093) debian-gnu-linux-vm sched_switch:
    { cpu_id = 0 }, { pid = 16226, tid = 16226 }, { prev_comm = "ball"
    , prev_tid = 16226, prev_prio = 20, prev_state = 0, next_comm =
    "ball", next_tid = 16229, next_prio = 20 }
[12:20:15.327534039] (+0.000001278) debian-gnu-linux-vm sched_switch:
    { cpu_id = 0 }, { pid = 16226, tid = 16229 }, { prev_comm = "ball"
    , prev_tid = 16229, prev_prio = 20, prev_state = 1, next_comm =
    "ball", next_tid = 16226, next_prio = 20 }
[12:20:15.328431490] (+0.000897451) debian-gnu-linux-vm sched_switch:
    { cpu_id = 0 }, { pid = 16226, tid = 16226 }, { prev_comm = "ball"
    , prev_tid = 16226, prev_prio = 20, prev_state = 0, next_comm =
    "gnome-shell", next_tid = 25370, next_prio = 20 }
[12:20:15.328906837] (+0.000475347) debian-gnu-linux-vm sched_switch:
    { cpu_id = 0 }, { pid = 16226, tid = 16229 }, { prev_comm = "ball"
    ,prev_tid = 16229, prev_prio = 20, prev_state = 1, next_comm =
    "Xorg", next_tid = 814, next_prio = 20 }
[12:20:15.329958090] (+0.001051253) debian-gnu-linux-vm sched_switch:
    { cpu_id = 0 }, { pid = 16226, tid = 16226 }, { prev_comm = "ball"
    , prev_tid = 16226, prev_prio = 20, prev_state = 0, next_comm =
    "ball", next_tid = 16229, next_prio = 20 }
[12:20:15.329962039] (+0.000003949) debian-gnu-linux-vm sched_switch:
    { cpu_id = 0 }, { pid = 16226, tid = 16229 }, { prev_comm = "ball"
    , prev_tid = 16229, prev_prio = 20, prev_state = 1, next_comm =
    "ball", next_tid = 16226, next_prio = 20 }
[12:20:15.330616044] (+0.000654005) debian-gnu-linux-vm sched_switch:
    { cpu_id = 0 }, { pid = 16226, tid = 16226 }, { prev_comm = "ball"
    , prev_tid = 16226, prev_prio = 20, prev_state = 0, next_comm =
    "Xorg", next_tid = 814, next_prio = 20 }
}
...
```

Cet exemple illustre une succession d'événements `sched_switch` d'une application temps-réel `ball`. Nous pouvons remarquer néanmoins que des préemptions se produisent entre `ball` et `Xorg`, `java`, `InputThread`, `gnome-shell`, `kworker/0:2`... Ce qui signifie que nous enregistrons des événements qui correspondent à d'autres actions de l'application qui n'ont rien à voir avec le système temps-réel, tel que l'affichage par exemple.

Également, pour pouvoir lancer cette commande il faudra avoir le pid de l'application. Ce qui veut dire que celle-ci devra être lancée avant que la session de trace ne démarre. Et dans ce cas nous ne pouvons obtenir les événements clone qui se produisent juste après le lancement de l'application sans imposer de contraintes à l'utilisateur.

Pour remédier à ce problème, au lieu de tracer un seul pid, le programme trace tous les processus présents sur la machine et les renvoie vers la commande `sed` qui se charge de filtrer les événements.

L'avantage de cette méthode et de pouvoir également proposer à l'utilisateur les événements qui se produisent en dehors de l'application temps-réel, ce qui permettrait de déterminer le degré d'implication des processus extérieurs à l'application. Ce qui signifie que le programme peut représenter non seulement les préemptions entre les tâches mais également celles entre les tâches temps-réel et les processus de la machine.

4-2-2- Parsing principal

Une fois cet première étape effectuée, il faut réussir à interpréter les traces texte. Pour cela, nous avons fait appel à **ANTLR4** (ANother Tool for Language Recognition), un puissant générateur d'analyseur pour la lecture, le traitement, l'exécution ou la traduction de fichiers texte structurés ou binaires. Il est largement utilisé pour construire des langages, des outils et des frameworks. A partir d'une grammaire, ANTLR génère un analyseur qui peut construire et parcourir des arbres parse.

C'est un outil java peut être utilisé pour récupérer les arguments de traces sous forme d'objets java. Dans notre cas, le fichier *parser* d'ANTLR reconnaît des traces sous la forme :

```
parser grammar traceParser;
options {tokenVocab=traceLexer;}


start returns [ List<trace> traces ] : (trace {...})* EOF;

trace returns [ trace tr ] :
    timestamp HOSTNAME ENTRY_CLONE COL LCBRA header RCBRA COM LCBRA
        context RCBRA COM LCBRA fields_entry_clone RCBRA { ... }
```

```
    | timestamp HOSTNAME EXIT_CLONE COL LCBRA header RCBRA COM LCBRA
        context RCBRA COM LCBRA fields_exit_clone RCBRA { ... }
    | timestamp HOSTNAME SCHED_SWITCH COL LCBRA header RCBRA COM LCBRA
        context RCBRA COM LCBRA fields_sched_switch RCBRA { ... }
    | timestamp HOSTNAME PTASK_TRACEPOINT COL LCBRA header RCBRA COM
        LCBRA fields_ptask_tracepoint RCBRA { ... }
    ;
```

...

Avec les quatre types de trace qui correspondent aux événements `entry_clone`, `exit_clone`, `sched_switch`, et le tracepoint de ptask. Cet arbre retourne par la suite une liste java de trace qui est reconnue par des class java.

En effet, un programme java a été réalisé en intégrant un package `parser` où sont générés les fichiers java correspondant au *parser* ANTLR, mais également une classe d'implémentation qui extrait la trace depuis le fichier texte, et crée une liste java de traces.

Les objets traces sont définis dans un autre package `event` avec plusieurs attributs selon que l'objet soit : `entry_clone`, `exit_clone`, `sched_switch`, ou le tracepoint de ptask.

4-3- Exploitations des données

Une fois la liste de traces obtenue, il faudra exploiter les informations extraites lors du traçage pour pouvoir rendre un résultat exploitable par les développeurs lors d'un debuggage.

Le résultat devra être de préférence graphique, accompagné de quelques chiffres et statiques qui correspondent à un résumé d'exécution (comme le nombre d'échéances manquées par exemple). Pour cela, plusieurs choix sont possibles.

La solution la plus évidente serait d'utiliser une API java pour les représentations graphiques. Plusieurs solutions *opensource* sont disponibles, tel que : **JFreeChart**.

Cette librairie graphique offre aux utilisateurs le maximum d'alternatives pour représenter leurs données complexes d'une manière simplifiée. Il permet l'utilisation extensive de diagrammes X-Y, diagrammes circulaires, diagrammes à barres, diagrammes de Gantt, diagrammes de Pareto, diagrammes combinés, diagrammes à plaquettes et autres types spéciaux de diagrammes.

Générer une courbe mathématique serait plus facile mais également compliqué à utiliser d'un point de vue utilisateur dans certains cas de systèmes temps-réel. Une autre solution serait de générer un fichier vectoriel tel que du `svg`.

Cette solution est beaucoup plus intéressante dans la mesure où elle permet de générer des fichiers plus légers ; l'image peut être plus aisément retravaillée (les

calques sont conservés) ; il n'y pas de compression de données avec perte, donc pas de détérioration de l'image ; et surtout, le redimensionnement de l'image ne génère aucune détérioration du dessin.

L'avantage également de cette solution est qu'elle peut être exploitée par plusieurs utilitaires, les fichiers peuvent être facilement convertis sous d'autre formats, et svg est un dérivé du xml qui fait partie des langages du web, ce qui fait qu'il peut être intégré facilement à du HTML.

Dans le programme java figure un autre package java `task`, qui contient des classes permettant de représenter une tâche et un événement temporel. Mais aussi une classe permettant de représenter l'activité de l'os et celle du thread principal qui lance les tâches.

Ce package intègre également une classe d'implémentation qui permet de convertir la liste de traces précédemment obtenues en liste de tâches. Elle permet également d'extraire une liste de classes java pour l'activité de chaque cpu et pour celle du *thread* principal de l'application. Chacun de ces objets contient des attributs qui permettent la reproduction de toute son exécution.

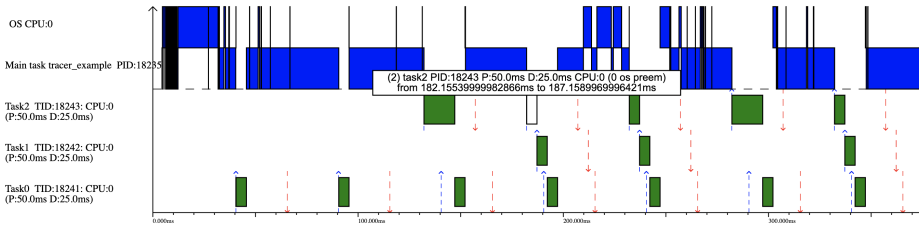
La classe d'implémentation comprend également d'autres méthodes qui permettent, entre autres, de vérifier si les événements de chaque tâche sont correctement chargés; de savoir si une exécution de tâche respecte la deadline fixée...

Après cette étape, et avec le package java `printer` du programme, il est possible de générer un fichier `svg` puis de représenter l'ordonnancement des tâches exécutées. Ce package utilise des objets `Graphics2D` pour la représentation du canvas puis des `Element DOM` java (*Document Object Model*). Cette différence réside dans le fait que pour créer des *tooltips* avec la première solution il est nécessaire de mettre des `handler` qui s'exécutent dynamiquement lorsque l'action précisée dans des méthodes `listener` est faite. Ce qui rend cette première solution compliquée à appliquer pour de la génération statique de codes.

Néanmoins pour d'autre versions, nous pouvons envisager de n'utiliser que des éléments `DOM` pour la representation graphique.

Le package `printer` utilise Batik, un *toolkit* basé sur Java pour les applications ou applets qui souhaitent utiliser des images au format SVG (Scalable Vector Graphics) à des fins diverses, comme l'affichage, la génération ou la manipulation.

La figure ci-dessous représente l'ordonnancement de l'exemple présent dans la partie **Introduction à ptask**, elle est générée par le programme java une fois que le traçage est fait :

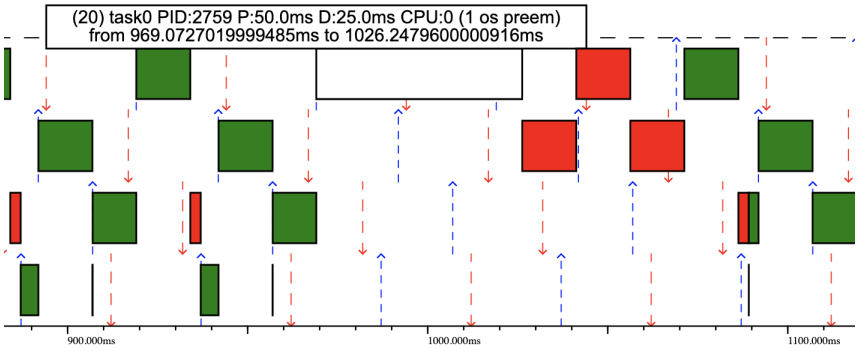


Une date de réveil est représentée avec une flèche bleu vers le haut, une deadline par une flèche rouge vers le bas. L'activité de l'os est sur la première ligne avec tous les processus qui s'exécutent sur la machine, le thread principal de l'application est représenté sur la ligne d'après, puis les tâches temps-réel sont représentées en vert s'ils respectent le deadline, sinon ils seront en rouge. Également des informations sur chaque exécution sont disponibles dans les *tooltips*.

4-4- Améliorations possibles

Plusieurs pistes d'améliorations restent possibles sur l'outil développé, parmi elles :

- Lors de l'utilisation normale de l'outil, l'activité de l'os est également tracée, ce qui limite la taille de la liste java de traces lors du chargement de la liste, mais également la taille de l'élément `root` qui contient les autres éléments du `svg` lors de la création de cet élément. L'une des pistes d'amélioration de l'outil serait de trouver un autre moyen de gestion de ces deux mémoires pour permettre d'avoir plus d'informations.
- Une autre alternative pour avoir une plus longue période de trace était de ne pas récupérer les traces issues d'un autre pid que celui de l'application tracée. Avec cette solution, nous ne savons pas quand est-ce que la tâche est préemptée par un autre processus



. L'indication (1 os preem) sert à indiquer qu'il y a bien préemption avec un autre processus de la machine, mais qu'il n'a pas été représenté. Une autre amélioration serait d'essayer de représenter cette préemption en ayant enregistré que les événements de l'application.

- Une fois la capacité mémoire de l'outil amélioré, il serait également intéressant d'adapter les canaux et buffers de LTTng (modifier la taille et le nombre des buffers, options de collecte des informations, ...) à l'utilisation de l'outil, de telle sorte à ne pas rater d'événements lorsque ces derniers sont trop rapprochés ou trop nombreux.
- Contrairement à la date de reveil d'une tâche périodique (flèche bleu) qui est obtenue avec la date du `clone`, celle de la tâche apériodique est obtenue grâce au tracepoint placé dans le programme (dans la fonction `wait_for_activation()`). Cette date dépend donc de l'exécution du programme et par conséquent n'est pas très précise. Pour améliorer cette précision il faudra donc trouver le moyen d'obtenir l'ordre d'activation de la tâche au lieu de détecter la date d'activation. (Piste : utiliser l'élément `at` de la structure `task_par`).

Conclusion

Pendant la première partie du projet a été réalisé un travail de recherche sur les utilitaires de traçage et sur l'API temps-réel ptask.

La deuxième partie du projet a servi à l'instrumentation de l'api et la réalisation du programme permettant le traçage du système temps-réel.

Ce travail a été très formateur et m'a permis de comprendre mieux les deux sujets, et surtout d'avoir une expérience dans le domaine de la programmation bas niveau qui m'intéresse particulièrement.

Durant la troisième partie du projet, le programme java qui effectue le *parsing*, puis analyse les traces pour produire un fichier `svg` a été réalisé.

Par la suite, dans un script shell ont étaient automatisées toutes les étapes nécessaires pour l'utilisation :

- Configuration de LTTng.
- Lancement de la session de trace.
- Lancement de l'application en arrière plan.
- Destruction de la session de trace et nettoyage.
- Enregistrement de la trace txt.
- Lancement du programme java.
- Enregistrement du fichier svg.

L'outil peut être lancé grâce à la commande :

```
ptaskTracer -f|--file [APP_NAME] <-p|--path APP_PATH> <-t|--time
    TRACING_TIME> <-s|--scale PRINT_SCALE> <-h|--theight TASK_HEIGTH>
    <-os|--osapp> <-g|--group> <-o|--output OUTPUT>
```

Ou, pour relancer avec une trace enregistrée et sans effectuer un autre traçage :

```
ptaskTracer --old -f|--file [APP_NAME] <-s|--scale PRINT_SCALE>
    <-h|--theight TASK_HEIGTH> <-os|--osapp> <-g|--group>
    <-i|--input INPUT>
```

Plus de details sur le répertoire git.

Un script d'installation a aussi été réalisé, il permet de :

- Installer LTTng.
- Télécharger les `jar` java nécessaire.
- Compiler le programme java.
- Générer un `jar` de l'application.

Sources

- Introduction à outil de trace LTTng.
- Introduction à outil de trace ftrace.
- Documentation officiel LTTng.
- Threads vs processus au niveau du kernel.
- Documentation officiel ptask.
- Bibliothèque lttng-ust.
- Format CTF.
- Manuel d'utilisation ptask.
- Documentation kernel tracepoint.
- Documentation kernel event.
- Manuel ANTLR
- Format SVG.
- Page de l'IRCICA
- Projet Batik