PENGEMBANGAN *PRIMARY-BACKUP SCHEDULING* PADA XEN ARINC 653

Laporan Tugas Akhir I

Disusun sebagai syarat kelulusan tingkat sarjana

IF4091/Tugas Akhir I dan Seminar

Oleh

AUFAR GILBRAN

NIM: 13513015

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

Januari 2017

PENGEMBANGAN *PRIMARY-BACKUP SCHEDULING* PADA XEN ARINC 653

Laporan Tugas Akhir I

Oleh

AUFAR GILBRAN

NIM: 13513015

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Bandung, 4 Januari 2017

Mengetahui,

Pembimbing,

Achmad Imam Kistijantoro, ST,M.Sc, Ph.D

NIP. 197308092006041001

DAFTAR ISI

DAFTAR ISI ..... **iii**

DAFTAR GAMBAR ..... **v**

BAB I. PENDAHULUAN **1**

I.1 Latar Belakang 1

I.2 Rumusan Masalah 3

I.3 Tujuan 4

I.4 Batasan Masalah 4

I.5 Metodologi 4

I.6 Sistematika Pembahasan 5

BAB II. STUDI LITERATUR **7**

II.1 ARINC 653 7

II.1.1 Sistem *Real-Time* 8

II.1.2 *Real-Time Scheduling* 10

II.1.3 Arsitektur Sistem 11

II.1.4 Fungsionalitas Sistem 12

II.2 *Primary-Backup Scheduling* 14

II.3 Virtualisasi 15

II.4 *Hypervisor* 16

II.5 Xen 17

II.5.1 Xen ARINC 653 18

BAB III. ANALISIS PERMASALAHAN DAN DESAIN SOLUSI **21**

III.1 Analisis Permasalahan 21

III.2 Analisis Solusi 22

III.2.1 Implementasi *Primary-Backup Scheduling* pada Xen ARINC 653 22

III.2.2 Pengujian dan Pengambilan Hasil 23

BAB IV. PENGEMBANGAN DAN PENGUJIAN..... 25

IV.1 Pengembangan *Primary-Backup Scheduling*..... 25

IV.1.1 Modul *scheduler* ARINC 653..... 26

IV.1.2 Modul *public hypercall*..... 27

IV.1.3 Modul libxc..... 27

IV.1.4 Modul libxl..... 27

IV.2 Pengujian..... 27

DAFTAR REFERENSI..... 28

BAB A. Fungsi `a653sched_do_schedule` 29

DAFTAR GAMBAR

Gambar II.1. Ilustrasi perbedaan permasalahan *scheduling* pada aplikasi *real-time* dan *non-real-time* 11

Gambar II.2. Contoh arsitektur *module* 12

Gambar II.3. Perbedaan penggunaan *ring native* dan *paravirtualized* 18

Gambar II.4. Jalur sebuah paket jaringan yang dikirim dari domU 19

Gambar II.5. *Hypervisor* Xen ARINC 653 20

Gambar II.6. *Schedule* partisi pada Xen ARINC 653 20

BAB I

PENDAHULUAN

I.1 Latar Belakang

Sistem penerbangan atau avionik adalah sistem yang terdiri dari perangkat elektronik yang digunakan pada pesawat terbang atau satelit. Avionik berfungsi untuk memberikan fasilitas untuk mengoperasikan pesawat terbang. Sistem yang termasuk dalam avionik adalah sistem komunikasi, navigasi, *dashboard* untuk manajemen sistem, dan sistem-sistem yang dipasang pada pesawat terbang untuk melakukan fungsionalitas tersendiri.

Avionik merupakan komponen penting dalam sebuah pesawat terbang dan merupakan *safety-critical system*, yaitu sistem yang apabila mengalami kegagalan dapat mengakibatkan kematian, kerugian besar, atau kerusakan lingkungan. Sistem operasi yang digunakan oleh avionik pada sebuah pesawat umumnya merupakan sistem *real-time*, yaitu sistem yang kebenaran komputasinya tidak hanya ditentukan oleh kebenaran hasil komputasi secara logika, tetapi juga waktu dimana hasil komputasi dapat digunakan (Shin and Ramanathan, 1994, p. 6). Maka, manajemen waktu sangat mendasar pada sistem *real-time*. Perangkat yang digunakan untuk avionik memiliki beberapa standar yang wajib dipenuhi. Standar tersebut dibuat oleh Aeronautical Radio, Incorporated (ARINC) dalam sebuah seri standar ARINC 600. Penelitian ini akan mendalami lebih lanjut mengenai sistem operasi *real-time* pada avionik yang dispesifikasikan pada penerbangan bernama ARINC 653.

Standar ARINC 653 menspesifikasikan partisi ruang dan waktu pada sistem operasi *real-time* agar avionik yang menggunakan sistem operasi tersebut *safety-critical*. Setiap proses aplikasi disebut sebagai partisi. Selain spesifikasi partisi, ARINC 653 juga mendefinisikan API untuk memudahkan manajemen partisi. Sistem operasi yang memenuhi standar ARINC 653 berfungsi untuk memanajemeni beberapa partisi beserta komunikasi antar partisi. Manajemen yang dilakukan oleh meliputi *resources* komputasi umum seperti CPU, *timing*, memori, dan *devices* serta *resources* untuk komunikasi. Penelitian ini akan mendalami lebih lanjut mengenai manajemen CPU, khususnya proses *scheduling*, yang dispesifikasikan oleh

standar ARINC 653.

Partisi pada standar ARINC 653 dapat membuat eksekusi dan pembaharuan perangkat lunak pada sistem tersebut lebih tepercaya dan fleksibel (Jin and Hyun-Wook, 2013). Partisi dapat diganti atau ditambahkan selama partisi tersebut lolos pengujian *scheduling*. Sebagai contoh, aplikasi navigasi yang dijalankan pada suatu partisi tertentu pada avionik dapat diganti dengan versi terbaru yang menggunakan algoritma navigasi yang lebih baik tanpa mengganggu partisi lain. Namun, standar ARINC 653 tidak menghasilkan sistem yang *fault-tolerant*. Akan terdapat kemungkinan terjadinya *fault* pada perangkat lunak selama sistem beroperasi, meskipun keamanan dan kebenaran perangkat lunak tersebut sudah terverifikasi. Hal tersebut akan mengakibatkan kegagalan sistem yang berakibat fatal karena sistem yang memenuhi standar ARINC 653 umumnya adalah *safety-critical system*.

Untuk menghindari permasalahan yang akan muncul akibat partisi *primary* yang gagal, *scheduler* dapat menjalankan partisi *backup* yang bersesuaian. Fungsionalitas yang diberikan partisi *backup* akan identik dengan fungsionalitas yang diberikan oleh partisi *primary*. Apabila partisi *primary* mengalami kegagalan ketika menjalankan *task*, partisi *backup* akan melindungi sistem dengan mengambil alih *task* milik partisi *primary*. Solusi ini dapat diimplementasikan dengan cara melakukan *scheduling* pada partisi *primary* terlebih dahulu. Jika *task* dari sebuah partisi *primay* gagal dikerjakan, maka *scheduler* akan menerima sinyal dan akan melakukan *scheduling* pada partisi *backup* yang bersesuaian. Hal tersebut mengakibatkan fungsi yang disediakan oleh aplikasi yang mengalami *fault* akan tetap tersedia dan sistem akan tetap berjalan sebagaimana seharusnya, sehingga sistem dapat dikatakan menjadi lebih *fault-tolerant*. Meski demikian, peningkatan tersebut hanya terlihat secara teoretis dan belum ada studi mengenai penggunaan *primary-backup scheduling* pada sistem ARINC 653. Untuk dapat melihat perbandingan antara *primary-backup scheduling* dengan *scheduling* normal, perlu dilakukan pengujian pada sistem ARINC 653 yang dapat menggunakan keduanya.

Arsitektur komputer yang dispesifikasikan pada standar ARINC 653 dapat divirtualisasikan dengan menggunakan *hypervisor*. *Hypervisor* adalah sebuah komponen yang membuat dan menjalankan *virtual machine*. Setiap *virtual machine* akan memiliki ruang memori dan waktu tersendiri sehingga dapat digunakan untuk mengimplementasikan sebuah partisi.

Salah satu solusi sistem ARINC 653 yang menggunakan *hypervisor* adalah Xen ARINC 653. Xen ARINC 653 adalah sebuah prototipe sistem ARINC 653 yang dibangun di atas Xen, yaitu *hypervisor open-source* dengan lisensi GPL sehingga Xen ARINC 653 juga *open-source*. Solusi *open-source* akan mempermudah pengembang aplikasi dan peneliti untuk melakukan pengembangan dan percobaan pada lingkungan ARINC 653 sebelum membeli solusi berbayar. Xen ARINC 653 merupakan solusi yang sangat tepat untuk melakukan perbandingan antara *primary-backup scheduling* dengan *scheduling* normal. Sayangnya, prototipe tersebut tidak menggunakan metode *primary-backup scheduling* sehingga perlu metode tersebut perlu diimplementasikan pada prototipe sebelum dapat dibandingkan.

I.2 Rumusan Masalah

Permasalahan yang terjadi berdasarkan uraian latar belakang adalah standar ARINC 653 tidak membuat sistem menjadi *fault-tolerant*. Padahal, fungsi dari standar ARINC 653 adalah sebagai panduan pengembangan sistem operasi *real-time* pada *safety-critical system*. Masalah tersebut dapat diselesaikan dengan menggunakan *primary-backup scheduling*. Meski *primary-backup scheduling* membuat sistem menjadi *fault-tolerant* secara teoretis, perlu adanya studi lebih lanjut apakah metode *primay-backup scheduling* dapat digunakan pada sistem ARINC 653. Selain itu, perlu adanya perbandingan antara *scheduling* normal dengan *primary-backup scheduling* pada ARINC 653 untuk melihat kemungkinan peningkatan atau penurunan efisiensi sistem.

Berikut adalah pertanyaan permasalahan yang akan dijawab dan diselesaikan pada tugas akhir ini:

- Bagaimana cara mengimplementasikan *primary-backup scheduling* pada Xen ARINC 653?
- Bagaimana cara melakukan pengujian *scheduling* pada Xen ARINC 653?
- Apakah penggunaan *primary-backup scheduling* pada sistem yang memenuhi standar ARINC 653 menghasilkan sistem yang lebih *fault-tolerant*?
- Apakah terdapat peningkatan atau penurunan kinerja akibat penggunaan *primary-backup scheduling* pada sistem yang memenuhi standar ARINC 653?

I.3 Tujuan

Berdasarkan uraian pada rumusan masalah, tujuan yang ingin dicapai adalah mengimplementasikan *primary-backup scheduling* pada Xen ARINC 653 beserta pengujian dan perbandingan hasil *scheduling* normal dengan *primary-backup scheduling*.

I.4 Batasan Masalah

Berdasarkan rumusan masalah yang ada, terdapat beberapa asumsi terkait dengan sistem avionik yang akan dikembangkan yaitu:

- Pengujian dan eksperimen yang dilakukan hanya akan diuji coba pada sistem dengan arsitektur x86-64.
- *Hypervisor* yang digunakan adalah Xen. Xen merupakan *hypervisor* tipe-1, sehingga mungkin berbeda dengan *hypervisor* tipe-2 karena *hypervisor* tipe-1 berjalan langsung di atas perangkat keras sedangkan *hypervisor* tipe-2 berjalan di atas sistem lain. Perbedaan ini mengakibatkan sistem yang dibangun di atas *hypervisor* tipe-1 tidak dapat berjalan pada *hypervisor* tipe-2. Penggunaan *hypervisor* tipe-1 lainnya mungkin akan memberikan perilaku yang berbeda dengan perilaku yang terjadi pada Xen.
- Pengujian dan eksperimen yang dilakukan hanya akan diuji coba dengan menggunakan *kernel* Linux untuk setiap partisi-nya. Hal ini mengakibatkan partisi yang tidak menggunakan *kernel* Linux mungkin memiliki perilaku yang berbeda.

I.5 Metodologi

Metodologi yang akan digunakan dalam penelitian tugas akhir adalah sebagai berikut:

1. Studi Literatur

   Pengerjaan tugas akhir diawali dengan mempelajari referensi berupa jurnal ilmiah dan dokumen resmi terkait dengan spesifikasi ARINC 653, arsitektur Xen, dan cara kerja Xen ARINC 653. Pembelajaran mengenai spesifikasi ARINC 653 dilakukan untuk dapat mengetahui kriteria penilaian yang akan digunakan sebagai patokan dalam pembuatan kerangka pengujian. Pembelajaran arsitektur Xen dilakukan untuk mengetahui

cara memodifikasi Xen ARINC 653 untuk mencari tahu bagaimana eksperimen pada prototipe tersebut akan dilakukan.

2. Analisis

   Pada tahap ini dilakukan analisis permasalahan yang berkaitan dengan topik tugas akhir ini. Analisis permasalahan akan membahas spesifikasi ARINC 653 dan arsitektur Xen untuk kemudian dijadikan sebagai solusi terhadap permasalahan.

3. Perancangan Solusi dan Pengujian

   Pada tahap ini dilakukan perancangan pengujian serta eksperimen yang akan dilakukan yang dapat menyelesaikan masalah-masalah yang telah dijelaskan pada Subbab I.4.

4. Pengembangan dan Pengujian

   Pada tahap ini dilakukan pembangunan sistem pengujian serta pengembangan modul eksperimen pada Xen ARINC 653.

5. Kesimpulan dan Saran

   Pada tahap ini dilakukan pengujian dengan metodologi pengujian yang dibangun dari tahap pembangunan solusi dan pengujian. Selanjutnya dilakukan pengujian hasil pengujian dan penarikan kesimpulan.

I.6 Sistematika Pembahasan

Struktur laporan tugas akhir ini adalah sebagai berikut.

1. Bab I Pendahuluan : Bab ini berisi latar belakang, masalah yang dibahas, tujuan, batasan masalah, metodologi, dan sistematika pembahasan tugas akhir.
2. Bab II Studi Literatur : Bab ini berisi dasar teori yang digunakan pada tugas akhir.
3. Bab III Analisis Masalah dan Perancangan Solusi : Bab ini berisi analisis terhadap permasalahan yang telah dipaparkan pada Pendahuluan serta perancangan solusi yang dapat menyelesaikan permasalahan tersebut.
4. Bab IV Pengembangan dan Pengujian : Bab ini menjelaskan pembangunan solusi pada Xen ARINC 653 secara rinci.

5. Bab V Kesimpulan dan Saran : Bab ini berisi kesimpulan dari hasil pengujian dan saran untuk penelitian selanjutnya.

BAB II

STUDI LITERATUR

II.1 ARINC 653

Standar ARINC 653 menspesifikasikan partisi ruang dan waktu pada sistem operasi *real-time* avionik. Avionik adalah sebuah perangkat elektronik yang berfungsi untuk memberikan fasilitas untuk mengoperasikan pesawat terbang. Avionik umumnya merupakan sistem *safety-critical*, yang berarti kegagalan pada sistem dapat mengakibatkan:

- Kematian atau cedera serius pada manusia
- Hilangnya atau rusaknya perlengkapan atau properti
- Kerusakan lingkungan

Pada avionik, *Integrated Modular Avionics* (IMA) merupakan sebuah konsep yang bertujuan memperkecil penggunaan daya dan meringankan perangkat (Garside and Pighetti, 2009, p. 2.A.2-1). Dalam arsitektur IMA, fungsionalitas avionik yang diharapkan menggunakan *resources* yang sama. Arsitektur IMA melakukan optimasi dengan cara memberikan *resource* seminimum mungkin pada masing-masing fungsionalitas avionik. Arsitektur ini memungkinkan eksekusi beberapa aplikasi avionik pada sebuah komputer yang sama. Hal tersebut dapat dicapai apabila sistem melakukan mekanisme partisi aplikasi avionik untuk mengisolasi eksekusi antar aplikasi, sehingga menjamin aplikasi dari satu partisi tidak dapat mengubah aplikasi, *devices*, aktuator, maupun data pada partisi lain (Rushby, 2000, pp. 11-12). Pada IMA, keseluruhan sistem disebut sebagai *integrated module* yang terdiri dari *core module* dan aplikasi. Sebuah *core module* terdiri atas *core software* dan *core hardware* yang menyediakan fungsionalitas *core module* dari sisi perangkat lunak dan perangkat keras. *Core module* berfungsi untuk menjadi tempat eksekusi aplikasi dan menyediakan layanan untuk melakukan partisi. Pada buku ini, istilah *integrated module* dan *core software* dapat dipertukarkan dengan *module* dan sistem operasi.

Standar ARINC 653 bertujuan untuk mengurangi biaya pengembangan, ukuran sistem, dan

biaya sertifikasi dengan cara menggabungkan beberapa aplikasi pada satu komputer namun tetap menjaga aplikasi-aplikasi tersebut saling terisolasi (Airlines Electronic Engineering Committee, 2012, pp. 3-30). Untuk mencapai tujuan tersebut, ARINC 653 mendefinisikan *interface* yang disebut sebagai *Application Executive* (APEX) untuk manajemen partisi, proses dan *timing*, komunikasi antar proses/partisi, dan penanganan *error* pada arsitektur IMA. Pada ARINC 653, satu unit partisi aplikasi disebut sebagai partisi.

Setiap partisi pada dasarnya sama seperti sistem dengan satu aplikasi karena partisi memiliki *device*, aktuator, konteks, dan data tersendiri. Hal tersebut mengakibatkan partisi ruang dan waktu yang kuat. Proses dalam sebuah partisi diperbolehkan untuk melakukan *multitasking*.

Perangkat lunak pada *platform* ARINC 653 meliputi:

- Partisi aplikasi avionik yang dispesifikasikan secara langsung oleh ARINC 653. Partisi aplikasi adalah target utama dari partisi ruang dan waktu dan dibatasi hanya menggunakan *system-call* yang telah disediakan oleh ARINC 653.
- Sebuah *kernel* sistem operasi yang menyediakan *interface* dan perilaku yang telah dispesifikasikan oleh standar ARINC 653. *Kernel* harus menyediakan lingkungan eksekusi standar untuk menjalankan aplikasi.
- Partisi sistem untuk melakukan fungsi di luar lingkup APEX. Fungsi yang dilakukan partisi sistem meliputi manajemen komunikasi antar *device* perangkat lunak maupun manajemen *fault*.
- Fungsi spesifik sistem seperti *device driver*, *debug*, dan *test*.

II.1.1 Sistem *Real-Time*

Sistem *real-time* adalah sistem yang kebenaran perhitungannya tidak hanya ditentukan oleh kebenaran hasil perhitungan secara logika, tetapi juga waktu dimana hasil perhitungan dapat digunakan (Shin and Ramanathan, 1994, pp. 6-7). Sistem *real-time* digunakan untuk mendukung aplikasi yang membutuhkan perhitungan *real-time*. Aplikasi yang membutuhkan perhitungan *real-time* umumnya terdiri atas beberapa *task* yang saling terkait. Sebuah *task* seringkali harus dikerjakan dalam interval reguler dan memiliki *deadline* yang berarti *task* tersebut harus sudah selesai pada waktu yang telah ditentukan.

Task demikian biasa disebut sebagai *periodic task*. Pada umumnya, *periodic task* merupakan *time-critical task* yang harus dijamin dapat diselesaikan sesuai dengan *deadline*-nya dan kegagalan dalam menyelesaikan *task* tersebut dapat mengakibatkan kegagalan sistem secara keseluruhan. Sebagai contoh, pada aplikasi kendali pesawat, sebuah *periodic task* mungkin melakukan pengaturan kekuatan mesin dengan cara menghitung jumlah bahan bakar yang harus diberikan dan memberikannya sejumlah hasil perhitungan pada mesin. Kegagalan sistem operasi menyelesaikan *task* tersebut dalam *deadline* yang telah ditentukan dapat mengakibatkan jatuhnya pesawat dan hilangnya nyawa manusia. Maka, sistem operasi *real-time* pada aplikasi kendali pesawat harus dapat memberikan jaminan bahwa *periodic task* aplikasi tersebut dapat diselesaikan sesuai dengan *deadline*-nya.

Tidak semua *task* harus dikerjakan dalam interval reguler. Beberapa *task* dikerjakan apabila suatu kejadian terjadi. *Task* demikian disebut dengan *aperiodic task*. Sebagai contoh, pintu pesawat dibuka dengan menekan tombol untuk membuka pintu pesawat. *Task* tersebut tidak memiliki *deadline* penyelesaian. Namun, *aperiodic task* juga dapat memiliki *deadline* seperti *periodic task*. Maka, *aperiodic task* demikian harus diselesaikan sebelum *deadline*-nya.

Berdasarkan penjelasan di atas, *deadline* dari sebuah sistem *real-time* dapat diklasifikasikan menjadi *hard*, *firm*, dan *soft*. Sebuah deadline dikatakan *hard* apabila konsekuensi kegagalan penyelesaiannya fatal. Pada umumnya, *periodic task* memiliki *hard deadline*. Sebuah deadline dikatakan *hard* apabila konsekuensi kegagalan penyelesaiannya fatal. Kebanyakan jenis *aperiodic task* yang memiliki deadline termasuk dalam kategori *firm deadline*, seperti transaksi pada sistem basis data (Haritsa, Carey, and Livny, 1992, pp. 203-241).

Penentuan *deadline* suatu aplikasi datang dari aplikasi itu sendiri. Sebagai contoh, pada sistem pengaturan kekuatan mesin, kekuatan mesin harus dapat diregulasi setiap 50 ms. *Deadline* tersebut menjatuhkan *deadline* pada *subtask* pengaturan kekuatan mesin seperti perhitungan jumlah bahan bakar yang dibutuhkan dan pemberian bahan bakar. Sebagai contoh, perhitungan jumlah bahan bakar harus diselesaikan dalam waktu 5 ms dan pemberian bahan bakar harus diselesaikan dalam waktu 20 ms. *Deadline* tersebut akan menjatuhkan *deadline* pada *subtask* masing-masing, dan seterusnya.

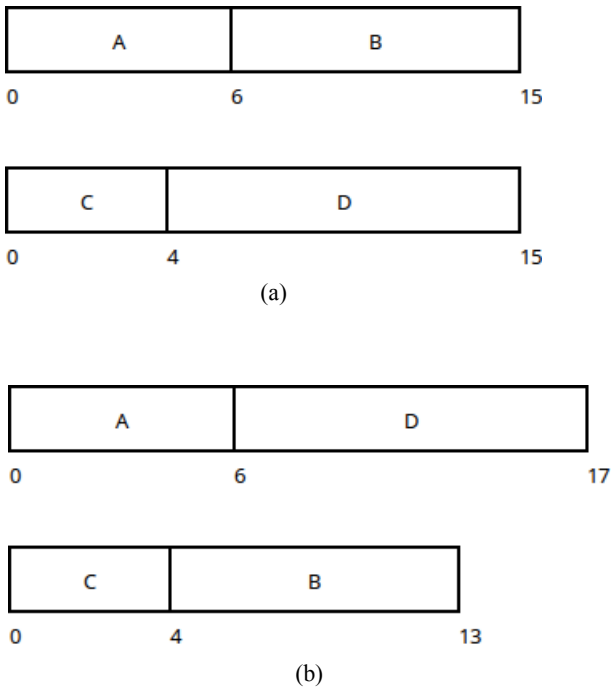
Dari gambaran di atas, karena pengaturan kekuatan mesin harus dapat diselesaikan tanpa terkecuali, terlihat bahwa sebuah sistem *real-time* harus dapat diprediksi. Sebuah sistem *real-time* dikatakan dapat diprediksi apabila pada saat desain dapat ditunjukkan bahwa seluruh *deadline task* aplikasi dapat dipenuhi selama beberapa asumsi mengenai sistem tersebut terpenuhi. Sebagai contoh, apabila asumsi mengenai sistem adalah *disk fault* hanya terjadi maksimal sebanyak f selama t detik. Maka, sistem tersebut dapat diprediksi apabila dapat ditunjukkan pada saat desain bahwa seluruh *deadline task* aplikasi dapat dipenuhi selama *disk fault* yang terjadi tidak lebih dari f selama t detik. Asumsi tersebut mengakibatkan penentuan sebuah sistem dapat diprediksi ditentukan oleh aplikasi yang dijalankan pada sistem tersebut.

II.1.2 *Real-Time Scheduling*

Diberikan beberapa *real-time task* dan *resouces* yang terdapat pada sistem, *real-time scheduling* adalah proses menentukan kapan dan dimana *task* tersebut akan dikerjakan untuk menjamin Shin and Ramanathan, 1994, pp. 8-9. *Scheduler* adalah sebuah proses yang melakukan *scheduling* pada *real-time task* yang diberikan. Dalam buku ini, istilah *real-time scheduling* dan *real-time scheduler* dapat dipertukarkan dengan *scheduling* dan *scheduler* sesuai dengan urutannya. Sistem operasi akan secara periodik memberikan sejumlah *real-time task* yang tersimpan pada memori pada *scheduler*. Pada aplikasi *non-real-time*, tujuan utama *scheduling* adalah untuk meminimisasi total waktu yang dibutuhkan untuk menyelesaikan semua *task*. Sementara itu, pada aplikasi *real-time*, tujuan utama *scheduling* adalah untuk memenuhi *deadline task* aplikasi.

Sebagai contoh, Gambar II.1(a) dan (b) memperlihatkan perbedaan antara *scheduling* pada sistem *real-time* dengan *scheduling* pada sistem *non-real-time*. *Task* A membutuhkan waktu 6 µs, *task* B membutuhkan waktu 9 µs, *task* C membutuhkan waktu 4 µs, dan *task* D membutuhkan waktu 11 µs. Seluruh *task* tersebut akan dijalankan pada sistem dengan dua buah CPU. Apabila sistem merupakan sistem *non-real-time*, maka *scheduling* yang terbaik dapat dicapai dengan pemilihan urutan proses seperti Gambar II.1(a). *Scheduling* seperti pada Gambar II.1(b) juga dapat digunakan, namun tidak optimal. Namun, apabila sistem merupakan sistem *real-time*, dan *task* B dan D mempunyai *deadline* 14 µs dan 17 µs, maka satu-

satunya *scheduling* yang memenuhi adalah seperti pada Gambar II.1(b).



Gambar II.1: Ilustrasi perbedaan permasalahan *scheduling* pada aplikasi *real-time* dan *non-real-time*. (a) *Scheduling* aplikasi *non-real-time* (b) *Scheduling* aplikasi *real-time*

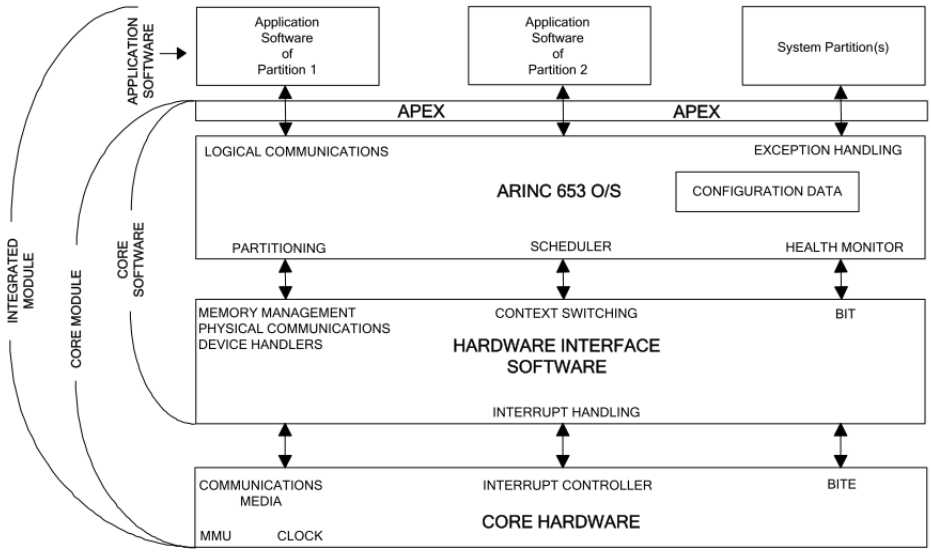
Algoritma untuk *scheduling* dapat diklasifikasikan dalam banyak dimensi. Beberapa algoritma *scheduling* dibuat untuk *periodic task* sedangkan algoritma lain dibuat untuk *aperiodic task*. Demikian juga beberapa algoritma dibuat untuk *preemptible task* sedangkan algoritma lain dibuat untuk *nonpreemptible task*. Algoritma *scheduling* dipengaruhi oleh faktor-faktor seperti tingkat kekritisan, keterkaitan, *resource*, dan *deadline* aplikasi.

Algoritma *scheduling* juga berbeda tergantung komputer tempat sistem akan dijalankan. Beberapa faktor komputer yang memengaruhi algoritma *scheduling* adalah jenis *processor* (*uniprocessor* atau *multiprocessor*), metode komunikasi *shared memory* atau *message-passing*, dan jaringan.

II.1.3 Arsitektur Sistem

Standar ARINC 653 menspesifikasikan arsitektur sistem menggunakan arsitektur IMA. Setiap *core module* dapat memiliki satu atau lebih *processor*. Arsitektur dari *core module* memengaruhi implementasi sistem operasi, namun tidak memengaruhi *interface* APEX yang

digunakan oleh *aplikasi* pada masing-masing partisi. Aplikasi harus mudah dipindahkan antar *core module* dan antar *processor* dari sebuah *core module* tanpa perlu adanya modifikasi *interface* dengan sistem operasi. Gambar II.2 menunjukkan contoh arsitektur sistem ARINC 653 serta komunikasi antar komponen pada sistem.



Gambar II.2: Contoh arsitektur *module* (Airlines Electronic Engineering Committee, 2012)

II.1.4 Fungsionalitas Sistem

Pada tingkat *core module*, sistem operasi memanajemeni partisi dan komunikasi antar partisi, yang mana mungkin dilakukan di dalam maupun antar *module*. Pada tingkat partisi, sistem operasi memanajemeni proses aplikasi dalam sebuah partisi dan komunikasi antar proses dalam partisi tersebut. Sistem operasi menyediakan fasilitas pada kedua level tersebut, namun memiliki perbedaan pada fungsi dan konten tergantung pada lingkup operasinya. Maka, definisi dari fasilitas yang diberikan sistem operasi membedakan pada tingkat apa fasilitas tersebut beroperasi.

Pada setiap saat, sebuah *module* berada dalam keadaan inisialisasi, operasional, atau diam. Pada saat dinyalakan, *module* memulai eksekusinya dalam keadaan inisialisasi. Setelah inisialisasi berhasil, *module* masuk ke dalam keadaan operasional. Selama *module* dalam keadaan operasional, sistem operasi memanajemeni partisi, proses aplikasi, dan komunikasi. *Module* akan terus berada pada keadaan operasional sampai dimatikan atau fungsi *health*

monitoring pada *module* memerintahkan *module* untuk masuk ke dalam keadaan inisialisasi kembali atau keadaan diam.

Berikut adalah fungsionalitas sistem yang dispesifikasikan oleh standar ARINC 653:

- Manajemen Partisi
- Manajemen Proses
- Manajemen Waktu
- Manajemen Memori
- Komunikasi Antar Partisi
- Komunikasi Dalam Partisi

Penelitian ini hanya akan membahas manajemen partisi dan manajemen waktu yang dilakukan oleh ARINC 653.

II.1.4.1 Manajemen Partisi

Inti dari standar ARINC 653 adalah konsep partisi, dimana aplikasi yang beroperasi dalam sebuah *module* dipartisi terhadap ruang (partisi memori) dan waktu. Sebuah partisi dapat dikatakan sebagai satu unit aplikasi yang didesain untuk memenuhi batasan-batasan partisi.

Sistem operasi pada dasarnya sudah cara untuk melakukan partisi. Sistem dengan partisi yang kuat memungkinkan partisi dengan tingkat kekritisan yang berbeda berjalan pada satu *module* yang sama tanpa memengaruhi partisi lain secara *ruang* dan waktu. Bagian dari sistem operasi yang bekerja pada tingkat *core module* bertanggung jawab untuk memaksakan partisi dilakukan dan manajemen partisi dalam *module* tersebut.

Partisi ditentukan urutannya dengan melakukan *scheduling* secara melingkar dan tetap. Untuk dapat melakukan *scheduling* secara melingkar, sistem operasi menyimpan *major time frame* untuk durasi tertentu yang kemudian akan diulang secara terus menerus selama *module* berjalan. Partisi akan terdaftar pada *major time frame* sebagai *partition window* yang memiliki nilai *offset* sesuai dengan urutan yang telah ditentukan. Partisi dengan dijalankan terlebih dahulu akan mendaftarkan *partition window* dengan nilai *offset* yang lebih kecil.

Urutan yang didapatkan oleh suatu partisi ditentukan oleh pengaturan urutan yang dibuat pada saat integrasi. Metode ini menjamin bahwa *scheduling* yang dilakukan deterministik dan dapat diprediksi.

Sebuah *module* dapat memiliki beberapa partisi yang memiliki periode pengerjaan yang berbeda-beda. *Major time frame* didefinisikan sebagai kelipatan dari kelipatan persekutuan terkecil dari seluruh periode pengerjaan partisi pada *module* tersebut. Setiap *major time frame* mempunyai partisi yang sama. Periode pengerjaan masing-masing partisi harus terpenuhi dengan mengatur frekuensi dan ukuran *partition window* dalam *major time frame*.

II.1.4.2 Manajemen Waktu

Manajemen waktu adalah proses penting yang menjadi karakteristik sebuah sistem operasi yang digunakan untuk sistem *real-time*. Waktu harus unik dan tidak bergantung pada eksekusi partisi apa pun dalam *module*. Seluruh nilai waktu yang digunakan harus terkait dengan waktu unik tersebut. Sistem operasi menyediakan *time-slicing* untuk *scheduling*, *deadline*, *periodicity* dari sebuah partisi.

Sebuah *time capacity* terasosiasi dengan setiap proses. *Time capacity* merepresentasikan waktu maksimal hasil perhitungan proses tersebut dapat digunakan untuk memenuhi syarat proses tersebut telah berhasil dikerjakan. Selama proses tersebut selesai dikerjakan tanpa melebihi *time capacity*-nya, *deadline* proses tersebut terpenuhi. Jika tidak, maka *deadline* proses tersebut tidak terpenuhi dan sistem akan menghasilkan *error*. *Error* hasil *deadline* yang tidak terpenuhi menjadi acuan bahwa sistem tidak berhasil menjamin 100% *realtime*.

II.2 *Primary-Backup Scheduling*

Studi untuk membuat sistem *real-time* menjadi *fault-tolerant* telah banyak dilakukan (Campbell, 1986) (Bertossi, Mancini, and Menapace, 2006). Campbell (1986) menunjukkan penambahan *deadline* pada setiap *task* akan menghasilkan sistem yang *fault-tolerant*. Meski demikian, penambahan *deadline* hanya menjamin waktu *response* sistem terhadap permintaan aplikasi dan hanya berfungsi sebagai indikator pada saat melakukan verifikasi sistem. *Fault* yang dialami aplikasi tidak hanya berupa waktu *response* saja, sehingga metode tersebut hanya membuat sistem menjadi *fault-tolerant* secara parsial. Untuk dapat mengatasi

fault yang tidak terduga pada saat verifikasi maupun *fault* pada proses verifikasi itu sendiri, salah satu solusi yang diusulkan adalah dengan menggunakan *primary-backup scheduling*.

Primary-backup scheduling merupakan proses *scheduling* yang terinspirasi oleh metode replikasi data. Pendekatan paling sederhana untuk melakukan *primary-backup scheduling* adalah dengan memberikan salinan untuk setiap *task*, dan memberikan salinan tersebut pada *processor* yang berbeda dengan *task* aslinya. Namun, pendekatan tersebut tidak efisien karena memerlukan banyak *processor* tambahan (Oh and Son, 1994). Bertossi, Mancini, and Menapace (2006) menunjukkan bagaimana cara mengimplementasikan *primary-backup scheduling* yang lebih efisien pada sistem *real-time* yang menggunakan algoritma *scheduling Rate- Monotonic-First-Fit* (RMFF). Implementasi tersebut memberikan *task salinan* pada *scheduler* jika dan hanya jika *task* utama mengalami kegagalan. Jin and Hyun-Wook (2013) berhasil membuktikan bahwa penggunaan *primary-backup scheduling* dapat dilakukan pada sistem yang terpartisi dengan melakukan perluasan model *resource*.

II.3 Virtualisasi

Dalam ranah komputasi, virtualisasi adalah tindakan untuk membuat versi virtual dari suatu hal. Virtualisasi pertama kali digunakan sekitar tahun 1960 sebagai metode untuk membagi *resources* sebuah komputer secara logika kepada beberapa aplikasi yang akan menggunakannya. Sejak saat itu, arti dari istilah virtualisasi mengalami perluasan makna.

Virtualisasi dapat dilakukan pada berbagai macam hal terkait dengan komputer. Sebagai contoh, sistem operasi modern sudah menggunakan konsep virtualisasi sederhana. Sebuah sistem operasi dapat mengatasi banyak aplikasi yang berjalan pada saat yang bersamaan yang mana masing-masing membutuhkan *resources* agar dapat berjalan sebagaimana seharusnya. Dalam kasus ini, sistem operasi memberikan ilusi kepada masing-masing proses bahwa proses tersebut memiliki seluruh *resources* yang terdapat pada komputer tersebut. Namun, yang terjadi adalah sistem operasi membagi *resources* komputer dengan metode sedemikian rupa sehingga pemakaian *resources* tetap konsisten dengan ilusi yang diberikan oleh sistem operasi kepada masing-masing proses.

Sebagai contoh, virtualisasi CPU dilakukan dengan melakukan *preemption* pada proses yang sedang menggunakan CPU sehingga memberikan kesempatan bagi proses lain untuk meng-

gunakan CPU. Sistem operasi melakukan melakukan *context switch* pada proses yang terkena *preemption* sehingga pada proses saat tersebut mendapatkan CPU pada giliran selanjutnya, proses tersebut dapat melanjutkan apa yang dikerjakan sebelumnya. Sementara itu, virtualisasi memori yang dilakukan dengan melakukan pemetaan alamat memori virtual yang mampu memiliki kapasitas 2^{32} (2^{64} pada sistem 64-bit) ke alamat memori fisik yang memiliki kapasitas sesuai dengan perangkat keras komputer. Kapasitas yang dapat digunakan oleh sebuah proses pada alamat memori fisik serta cara pemetaannya dilakukan oleh perangkat MMU pada komputer. *Devices* yang dimiliki oleh sebuah komputer juga divirtualisasikan oleh sistem operasi. Contoh *device* yang divirtualisasikan oleh sistem operasi adalah *storage*.

Salah satu kegunaan metode virtualisasi pada ranah komputasi adalah untuk membuat *virtual machine*. Sebuah *virtual machine* adalah representasi mesin secara logika yang memiliki *resources* layaknya mesin biasa. Umumnya, *virtual machine* digunakan untuk memberikan ilusi sebuah komputer sehingga dari sudut pandang perangkat lunak yang berjalan di atasnya, perangkat lunak tersebut seakan berjalan satu buah komputer fisik tanpa ada pembagian *resources*. Tentunya hal tersebut hanya berlaku untuk sistem operasi atau perangkat lunak yang berjalan langsung di atas perangkat keras. Sebuah komputer dapat direpresentasikan menjadi beberapa *virtual machine* sehingga memungkinkan beberapa sistem operasi beroperasi secara bersamaan pada komputer tersebut.

II.4 *Hypervisor*

Virtualisasi dilakukan menggunakan perangkat lunak virtualisasi yang disebut sebagai *hypervisor*. *Hypervisor* adalah sebuah perangkat lunak yang bertugas untuk membuat dan menjalankan beberapa *virtual machine*. *Hypervisor* berfungsi untuk memanajemeni *resources* fisik yang tersedia untuk digunakan oleh *virtual machine* yang berjalan di bawah *hypervisor* tersebut. Dengan demikian, *hypervisor* memiliki peran sama seperti sistem operasi, hanya saja ketimbang memanajemeni *resources* fisik untuk digunakan oleh proses perangkat lunak, *hypervisor* memanajemeni *resources* untuk sebuah proses yang juga mengatur *resources* yang diberikan pada proses tersebut untuk proses perangkat lunak yang berjalan di bawahnya. Istilah *hypervisor* didapatkan karena *hypervisor* membawahkan *supervisor*,

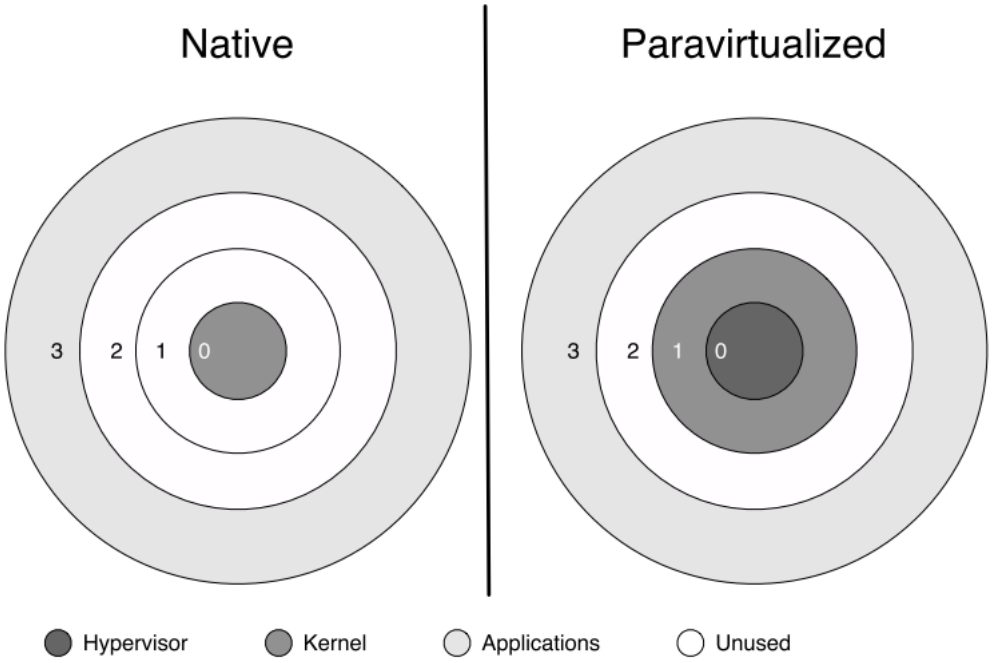
pada kasus ini *supervisor*-nya merupakan sistem operasi.

II.5 Xen

Xen merupakan sebuah perangkat lunak untuk melakukan virtualisasi yang bekerja diantara sistem operasi dan perangkat keras. Xen menyediakan lingkungan virtual dimana sebuah *kernel* dapat beroperasi. Terdapat tiga komponen utama pada sistem yang menggunakan Xen, yaitu *hypervisor*, *kernel*, dan aplikasi yang bekerja pada *userspace*.

Pada Xen, *kernel* sebuah sistem operasi tidak lagi beroperasi pada *ring* 0. *Ring* 0 digunakan oleh *hypervisor* yang akan mengatur *resources* yang akan diberikan pada *virtual machine* di bawahnya. Dimana *kernel* sistem operasi dijalankan bergantung pada arsitektur *processor*-nya. Sebagai contoh, pada sistem IA32, *kernel* sistem operasi berjalan pada *ring* 1 seperti pada Gambar II.3. Hal tersebut mengakibatkan *kernel* yang berjalan di atas Xen tetap dapat mengakses memori yang teralokasi untuk aplikasi yang berjalan di atas kernel tersebut, namun tetap tidak dapat diakses oleh aplikasi maupun *kernel* lain. *Hypervisor*, yang berjalan pada *ring* 0, terproteksi dari *kernel* yang berjalan pada *ring* 1 dan aplikasi yang berjalan pada *ring* 3. Karena *kernel* berjalan pada *ring* 1, *kernel* tidak dapat menjalankan *privileged instructions* seperti biasanya. *Hypervisor* pada Xen menyediakan *hypercall* sebagai pengganti *privileged instructions*.

Fungsi dari sebuah *hypervisor* adalah untuk menjalankan *virtual machine*. Pada Xen, *hypervisor* menjalankan *virtual machine* yang disebut sebagai *domain*. Terdapat dua jenis *domain* pada Xen, yaitu dom0 dan domU. Dom0 merupakan *domain* yang memiliki *privileged instructions* untuk memanajemeni *devices* yang dijalankan oleh Xen pada saat sistem pertama kali beroperasi. Kewajiban dom0 dalam menangani *devices* adalah dom0 harus dapat menyediakan *devices* untuk semua domU. Karena perangkat keras tidak dapat diakses oleh beberapa sistem operasi sekaligus, maka akses *devices* oleh domU harus melalui dom0. Gambar II.4 mengilustrasikan bagaimana paket jaringan yang dikirim oleh domU melalui dom0. Perbedaan utama pada pengiriman paket jaringan adalah pada *layer TCP/IP*, *layer* tersebut tidak meneruskan langsung ke *network device* tetapi menulis paket pada *shared memory*. Kemudian dom0 akan membaca paket yang terdapat pada *shared memory* untuk dilanjutkan ke *network device* yang sebenarnya. Penggunaan *shared memory* seperti ilus-



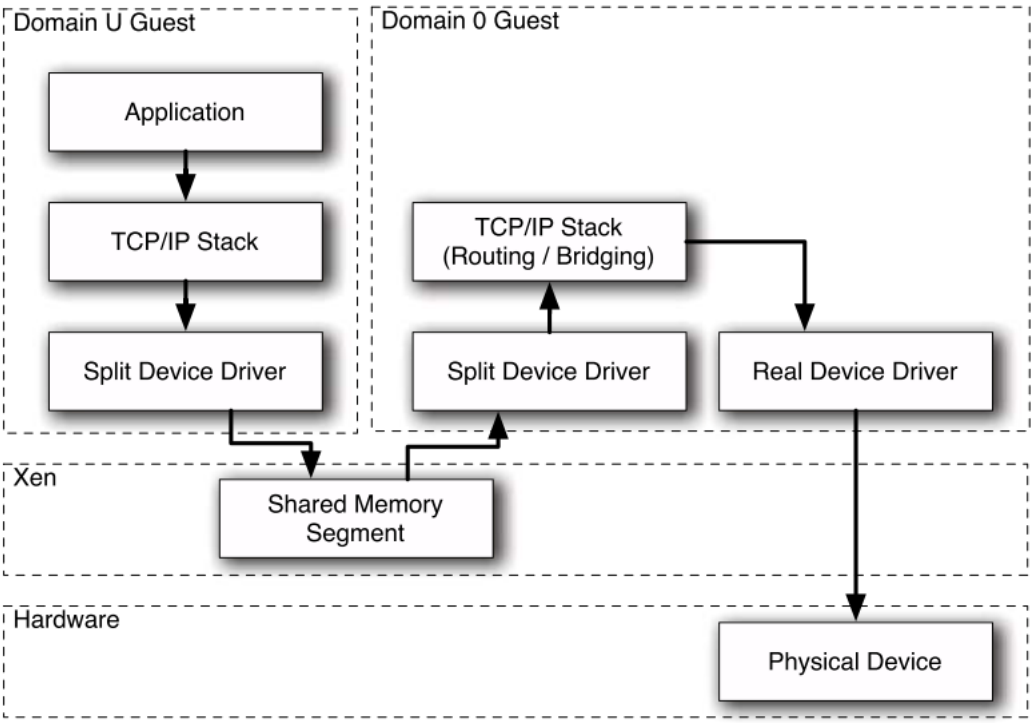
Gambar II.3: Perbedaan penggunaan *ring native* (kiri) dan *paravirtualized* (kanan) (Chisnall, 2014)

trasi tersebut banyak digunakan pada berbagai komponen yang membentuk Xen.

Salah satu kelebihan Xen sebagai *hypervisor* adalah Xen memudahkan pengembang untuk menuliskan komponen sesuai dengan kebutuhan. Xen menjelaskan secara rinci bagaimana proses penulisan komponen dan memberikan *interface* yang relatif mudah. Xen juga merupakan *hypervisor open-source*, sehingga eksplorasi keseluruhan sistem lebih mudah dilakukan. Fokus utama pada buku ini adalah, Xen menyediakan *interface* untuk membuat *scheduler* baru. *Scheduler* dalam kasus ini tidak bekerja pada proses, melainkan bekerja pada *domain*.

II.5.1 Xen ARINC 653

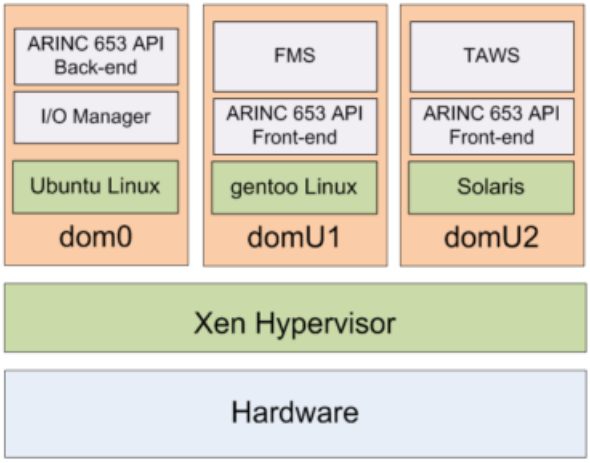
Xen ARINC 653 merupakan prototipe sistem ARINC 653 yang dibangun di atas Xen. Prototipe dibangun dengan membuat *scheduler* seperti pada spesifikasi standar ARINC 653 pada Xen. Selain itu, prototipe tersebut memberikan beberapa modul *kernel* untuk menangani *device* yang terpartisi sesuai dengan spesifikasi ARINC 653. Dalam melakukan partisi memori, prototipe tersebut memanfaatkan mekanisme partisi memori menggunakan MMU yang sudah merupakan mekanisme bawaan pada Xen.



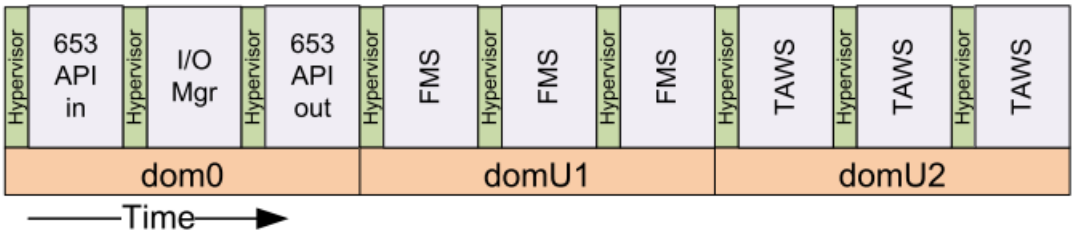
Gambar II.4: Jalur sebuah paket jaringan yang dikirim dari domU (Chisnall, 2014)

Setiap partisi akan direpresentasikan dengan menggunakan *domain* pada Xen. Dom0 pada Xen akan digunakan sebagai partisi tempat APEX dan *device driver* berada, sedangkan partisi untuk aplikasi avionik akan direpresentasikan menggunakan domU seperti pada Gambar II.5.

Scheduling dilakukan dengan metode seperti pada spesifikasi ARINC 653. *Task* sebuah aplikasi akan dimasukkan pada *major time frame* pada saat *scheduling*. Pada Xen, hanya dom0 yang dapat memproses IRQ untuk diteruskan kepada *hypervisor* atau *device driver*. Karena itu, *scheduling* yang dilakukan secara berkala akan dom0 pada *major time frame* seperti pada Gambar II.6.



Gambar II.5: *Hypervisor* Xen ARINC 653



Gambar II.6: *Schedule* partisi pada Xen ARINC 653

BAB III

ANALISIS PERMASALAHAN DAN DESAIN SOLUSI

III.1 Analisis Permasalahan

Standar ARINC 653 berfungsi sebagai panduan pengembangan sistem operasi *real-time* pada *safety-critical system*. Banyak faktor yang menentukan apakah sebuah sistem dapat dikategorikan sebagai *safety-critical system*, salah satunya adalah *fault-tolerancy*. Pada *safety-critical system*, *fault-tolerant* adalah properti yang sangat dicari. Sistem yang *fault-tolerant* akan memberikan jaminan bahwa sistem akan terus bekerja sebagaimana seharusnya meskipun terjadi *fault* selama sistem beroperasi.

Fokus dari standar ARINC 653 adalah menjamin sistem operasi melakukan partisi lingkungan eksekusi agar aplikasi yang berjalan pada sebuah lingkungan eksekusi tidak mengganggu aplikasi pada lingkungan eksekusi lainnya. Meski standar tersebut menjamin *fault* yang terjadi pada sebuah aplikasi tidak akan mengganggu aplikasi lain yang berada pada lingkungan eksekusi berbeda (*fault containment*), *fault* yang terjadi pada sebuah aplikasi tetap akan terjadi dan tidak ditangani. Standar ARINC 653 mendefinisikan mekanisme untuk mendeteksi dan memberikan tanggapan apabila terjadi *fault*. Namun, standar tersebut tidak mendefinisikan tanggapan apa yang harus dilakukan untuk menjamin aplikasi yang menghasilkan *fault* akan terus bekerja. Meskipun aplikasi pada sistem tersebut akan melalui proses verifikasi untuk menjamin tidak akan terjadi *fault*, proses verifikasi tersebut mungkin sangat lama atau memiliki *fault* juga. Karena itu, mekanisme penanganan *fault* tidak dapat diabaikan begitu saja.

Solusi dari permasalahan tersebut adalah *primary-backup scheduling*. Penggunaan metode *primary-backup scheduling* dapat menjamin bahwa aplikasi yang *fault* akan tetap menjalankan *backup* aplikasi tersebut yang sudah terverifikasi tidak akan mengalami *fault* dalam waktu panjang. Namun, meski sudah banyak studi mengenai *primary-backup scheduling* (Campbell, 1986) (Bertossi, Mancini, and Menapace, 2006), belum ada studi mengenai metode tersebut spesifik pada sistem yang memenuhi standar ARINC 653. Hal tersebut

mengakibatkan sulitnya pengembangan dan verifikasi *fault-tolerancy* sistem ARINC 653 yang menggunakan *primary-backup scheduling*.

III.2 Analisis Solusi

Eksperimen *primary-backup scheduling* akan dilakukan pada prototipe ARINC 653 pada Xen. Hal ini dikarenakan prototipe tersebut tersedia gratis dan *open-source*, sehingga pengembangan dan eksperimen yang akan dilakukan tidak memerlukan biaya banyak dan dapat dieksplorasi secara mandiri.

III.2.1 Implementasi *Primary-Backup Scheduling* pada Xen ARINC 653

Pada Xen, sebuah *domain* hanya dapat dibuat oleh dom0 saja. Aplikasi maupun *scheduler* tidak dapat membuat *domain* baru apabila aplikasi tersebut mengalami *fault*. Agar partisi *backup* dapat berjalan apabila partisi *primary* mengalami *fault*, maka partisi *backup* sudah harus dibuat oleh dom0 pada tahap inisialisasi. *Scheduler* akan memasukkan sebuah partisi pada *major time frame* pada tahap *scheduling* jika dan hanya jika partisi tersebut merupakan *primary* dan belum mengalami kegagalan. Jika terjadi kegagalan pada partisi *primary*, maka *scheduler* akan memasukkan partisi *backup* yang bersesuaian pada *major time frame*. Karena itu, untuk dapat mengimplementasikan *primary-backup scheduling* pada prototipe ARINC 653, *scheduler* harus dapat mengetahui partisi mana yang merupakan *primary* atau *backup* agar dapat melakukan *scheduling*.

Selain itu, *scheduler* juga perlu mengetahui kapan sebuah partisi *primary* dikategorikan mengalami kegagalan. Karena itu, perlu ada mekanisme untuk mendeteksi *fault*, dan memberikan informasi tersebut kepada *scheduler*.

III.2.1.1 Rancangan Solusi untuk Mengenali Jenis Partisi

Jenis partisi dapat dikenali dengan menambahkan properti *partition_type* pada konfigurasi *domain*, sehingga partisi juga mempunyai properti tersebut. Konfigurasi *domain* pada Xen dapat dilakukan dengan menggunakan *xl*, salah *tools* untuk melakukan manajemen *domain* yang tersedia pada *toolstack* milik Xen. Fungsi untuk melakukan pengaturan dapat ditambahkan dengan memanfaatkan XAPI, yaitu *interface* yang disediakan oleh Xen untuk

menambahkan fungsionalitas. Fungsionalitas yang akan ditambahkan adalah fungsionalitas untuk mendefinisikan *partition_type* pada *xl*. *Scheduler* akan dapat mendeteksi jenis *domain* melalui informasi tersebut.

III.2.1.2 Rancangan Solusi untuk Mendeteksi Kegagalan

Pada Xen, sebuah dom0 dapat mendefinisikan aksi yang harus dilakukan apabila sebuah *domain* mengalami *crash*. Salah satu aksi yang dapat dilakukan adalah menghapus *domain* tersebut, sehingga permasalahan apabila kegagalan yang dialami adalah *crash*, *domain* tersebut otomatis tidak akan diberikan pada *scheduler*. Namun, apabila permasalahan yang terjadi adalah *deadline* yang tidak terpenuhi, atau kesalahan yang tidak mengakibatkan *crash*, maka aplikasi harus mengirim IRQ kepada *hypervisor*. IRQ yang diberikan akan ditangani melalui dom0 yang dapat mengatur *domain*, sehingga IRQ tersebut dapat menjadi pertanda bahwa *domain* yang mengirimkan IRQ tersebut mengalami kegagalan dan meminta dom0 untuk menanganinya. Berikut beberapa penanganan yang dapat dilakukan oleh dom0:

- Menghapus *domain* yang mengirim IRQ tersebut
- Mengganti properti *domain* menjadi *temporary-primary*

Pemilihan penanganan akan ditentukan setelah studi lebih lanjut.

III.2.2 Pengujian dan Pengambilan Hasil

Pengujian *real-time* akan dilakukan dengan melakukan *stress-testing* menggunakan *scheduler* tertentu. *Scheduler* yang akan diuji adalah *scheduler* bawaan prototipe ARINC 653 dan *scheduler* implementasi *primary-backup scheduling*. *Stress-testing* dilakukan dengan menggunakan aplikasi yang akan menjalankan sebuah proses subjek dan menjalankan beberapa proses *background* (mungkin nol). Proses subjek akan memberikan laporan secara berkala pada aplikasi penguji apakah *deadline* dari *task* yang dibutuhkan terpenuhi atau tidak.

Pengujian *fault-tolerancy* akan dilakukan oleh aplikasi pengujian dengan cara memberikan *request* pada proses subjek, kemudian memvalidasi *response* yang diterima dari proses subjek. *Request* yang dilakukan harus sebuah pekerjaan yang dapat divalidasi kebenaran hasil-

nya oleh aplikasi penguji. Alur pengujian *fault-tolerancy* dilakukan sebagai berikut:

1. Aplikasi pengujian akan meminta proses subjek untuk menghitung suatu nilai berdasarkan masukan tertentu (misal menghitung nilai $\sqrt{n}$).
2. Proses subjek akan mengambil nilai semi-acak, berdasarkan nilai tersebut proses dapat melakukan salah satu dari:
    - Proses subjek akan menghitung nilai $\sqrt{n}$ dan memberikan *response* berupa hasil perhitungan.
    - Proses subjek akan mengaktivasikan mekanisme *fault detection*.
3. Aplikasi pengujian akan melakukan validasi kebenaran hasil perhitungan.

Untuk pengujian ini, aplikasi pengujian tidak akan menjalankan proses *background* sama sekali.

Kedua metode ini akan memberikan persentase *real-time* serta *fault-tolerancy scheduler* yang digunakan pada saat pengujian. Nilai semi-acak yang akan digunakan sebagai ambang batas keputusan proses subjek akan ditentukan sedemikian sehingga memiliki *mean time to failure* yang diharapkan. Hasil pengujian akan memberikan statistik faktor-faktor yang penting untuk sebuah *scheduler*.

BAB IV

PENGEMBANGAN DAN PENGUJIAN

IV.1 Pengembangan *Primary-Backup Scheduling*

Pengembangan dilakukan dengan memodifikasi kode *scheduler* ARINC 653 *source-tree* pada Xen ARINC 653. Modifikasi kode dilakukan pada modul-modul berikut:

- Libxl

  Modul ini digunakan untuk membuat *tools* yang dapat digunakan pada *user space domain* `dom0` untuk melakukan pengaturan *hypervisor*.

- Libxc

  Modul ini berisi fungsi-fungsi untuk menangani permintaan *hypercall* dari *domain*.

- *Public hypercall*

  Modul ini berisi definisi *interface* yang digunakan untuk mengatur struktur data yang dikirimkan pada saat mengirim data dari *domain* kepada *hypervisor*.

- *Scheduler* ARINC 653

  Modul ini berisi fungsi-fungsi yang digunakan untuk mengisi definisi *interface scheduler* pada Xen. Fungsi-fungsi tersebut meliputi algoritma *scheduler*, penanganan *hypercall* terhadap *scheduler*.

Agar algoritma *primary-backup scheduling* yang dijelaskan pada Subbab III.2 dapat berjalan dengan seharusnya, perlu ada mekanisme penanganan data yang dibutuhkan pada masing-masing domain. Hal tersebut dapat dicapai dengan mendefinisikan struktur data pada mekanisme pengiriman data dari *user space* kepada *hypervisor*, pendefinisian struktur data yang dimengerti oleh *hypervisor* dan *user space* untuk digunakan pada saat melakukan *hypercall*, mekanisme pengiriman data melalui *hypercall*, struktur data untuk menyimpan informasi *domain* pada *scheduler*, serta konversi data yang didapat oleh *hypervisor* setelah menangani *hypercall* menjadi dengan struktur data *domain* pada *scheduler*.

Implementasi mekanisme pengiriman data dari *user space* kepada *hypervisor* akan dipaparkan pada subsection IV.1.4. Mekanisme *hypercall* yang diimplementasikan akan dipaparkan pada subsection IV.1.3. Pendefinisian struktur data yang digunakan pada saat melakukan *hypercall* akan dipaparkan pada subsection IV.1.2. Mekanisme konversi data yang didapat oleh *hypervisor* setelah menangani *hypercall* serta definisi struktur data masing-masing *domain* pada *scheduler* akan dijelaskan pada subsection IV.1.1.

Penjelasan akan dilakukan secara *bottom-up* dimulai dari definisi *interface*/struktur data, kemudian fungsi dari *interface*/struktur data yang sudah didefinisikan. Masing-masing mekanisme akan dijelaskan apabila seluruh *interface*/struktur data yang digunakan pada mekanisme tersebut telah dipaparkan. Secara garis besar, penjelasan akan dilakukan dimulai dari bagian *low-level* dan *high-level* pada *hypervisor*, kemudian memasuki bagian *low-level* dan *high-level* pada *user space*. Urutan penjelasan dilakukan sedemikian dengan harapan akan mempermudah pembaca dalam mengerti masing-masing langkah yang akan dipaparkan pada setiap mekanisme.

IV.1.1 Modul *scheduler* ARINC 653

Pada Xen, pendefinisian *scheduler* dilakukan dengan membuat *interface scheduler*. *Interface* dibuat berdasarkan struktur yang telah terdefinisi pada `xen/include/xen/sched-if.h`.

```
struct scheduler {
    void        (*free_domdata) (const struct scheduler *, void *);
    void *      (*alloc_domdata) (const struct scheduler *, struct domain *);
    int         (*init_domain) (const struct scheduler *, struct domain *);
    void        (*destroy_domain) (const struct scheduler *, struct domain *);

    struct task_slice (*do_schedule) (const struct scheduler *, s_time_t,

    int         (*adjust)      (const struct scheduler *, struct domain *,
    int         (*adjust_global) (const struct scheduler *,
};
```

Listing IV.1: *Interface* dari fungsi *scheduler* yang dimodifikasi/diimplementasi

Setiap fungsi pada struktur tersebut akan berlaku sebagai *handler* yang akan dipanggil oleh *hypervisor* apabila terjadi *event* yang bersesuaian. Asosiasi antara *handler* dan *event* telah ditentukan sebelumnya dan dapat dilihat pada `xen/common/schedule.c`. Setiap *handler* pada struktur tersebut kemudian diisi dengan alamat dari fungsi yang nantinya akan menangani permintaan pada *scheduler*. Setiap *scheduler* diharuskan mengisi *handler* `do_-`

`schedule`. Apabila *scheduler* tidak memerlukan sebuah *handler*, maka *handler* dapat diisi dengan nilai `NULL`.

Untuk mengimplementasikan *scheduler*, fungsi `do_schedule` harus diisi dengan alamat dari fungsi yang akan melakukan *scheduling*. Pada implementasi *primary-backup scheduler*, fungsi `do_schedule` akan diisi dengan alamat dari fungsi `a653sched_do_schedule`. Definisi fungsi `a653sched_do_schedule` dapat dilihat pada Appendix A.

IV.1.2 Modul *public hypercall*

IV.1.3 Modul libxc

IV.1.4 Modul libxl

IV.2 Pengujian

DAFTAR REFERENSI

Airlines Electronic Engineering Committee (2012). “653P1-3 Avionics Application Software Standard Interface Part 1 - Required Services”. In: *Arinc 653*, p. 85.

Bertossi, Alan A., Luigi V. Mancini, and Alessandra Menapace (2006). “Scheduling hard-real-time tasks with backup phasing delay”. In: *Proceedings - IEEE International Symposium on Distributed Simulation and Real-Time Applications, DS-RT*, pp. 107–116. ISBN: 0769526977. DOI: 10.1109/DS-RT.2006.33.

Campbell, R O Y H (1986). “A Fault-Tolerant Scheduling Problem”. In: 11, pp. 1089–1095.

Chisnall, David (2014). *The Definitive Guide to the Xen Hypervisor*. Vol. 25. 9, p. 307. ISBN: 9780874216561. DOI: 10.1007/s13398-014-0173-7.2.

Garside, Richard and F. Joe Pighetti (2009). “Integrating modular avionics: A new role emerges”. In: *IEEE Aerospace and Electronic Systems Magazine* 24.3, pp. 31–34. ISSN: 08858985. DOI: 10.1109/MAES.2009.4811086.

Haritsa, Jayant R., Michael J. Carey, and Miron Livny (1992). “Data access scheduling in firm real-time database systems”. In: *Real-Time Systems* 4.3, pp. 203–241. ISSN: 09226443. DOI: 10.1007/BF00365312.

Jin, Hyun-Wook and Hyun-Wook (2013). “Fault-tolerant hierarchical real-time scheduling with backup partitions on single processor”. In: *ACM SIGBED Review* 10.4, pp. 25–28. ISSN: 15513688. DOI: 10.1145/2583687.2583693.

Oh, Yingfeng and Sang H Son (1994). “Enhancing Fault-Tolerance in Rate-Monotonic Scheduling”. In: 329, pp. 315–329.

Rushby, John (2000). “Partitioning in avionics architectures: requirements, mechanisms and assurance”. In: *Work* March, p. 67. DOI: DOT/FAA/AR-99/58.

Shin, Kang G. and Parameswaran Ramanathan (1994). “Real-Time Computing: A New Discipline of Computer Science and Engineering”. In: *Proceedings of the IEEE* 82.1, pp. 6–24. ISSN: 15582256. DOI: 10.1109/5.259423.

BAB A

Fungsi a653sched_do_schedule

```
static struct task_slice
a653sched_do_schedule(
    const struct scheduler *ops,
    s_time_t now,
    bool_t tasklet_work_scheduled)
{
    struct task_slice ret;               /* hold the chosen domain */
    struct vcpu * new_task = NULL;
    static unsigned int sched_index = 0;
    static s_time_t next_switch_time;
    a653sched_priv_t *sched_priv = SCHED_PRIV(ops);
    const unsigned int cpu = smp_processor_id();
    static bool task_done[ARINC653_MAX_DOMAINS_PER_SCHEDULE] = {0};
    static a653sched_domain_t * dom_priv = NULL;
    unsigned long flags;

    spin_lock_irqsave(&sched_priv->lock, flags);

    if ( sched_priv->num_schedule_entries < 1 )
        sched_priv->next_major_frame = now + DEFAULT_TIMESLICE;
    else if ( now >= sched_priv->next_major_frame )
    {
        /* time to enter a new major frame
         * the first time this function is called, this will be true */
        /* start with the first domain in the schedule */
        sched_index = 0;
        sched_priv->next_major_frame = now + sched_priv->major_frame;
        next_switch_time = now + sched_priv->schedule[0].runtime;
        dom_priv = DOM_PRIV(sched_priv->schedule[0].vc->domain);
        memset(task_done, 0, sizeof(task_done));
    }
    else
    {
        /* When switching domain, the dom_priv should point to current domain private data
             structure */
        BUG_ON(dom_priv == NULL);

        /* if last task elapsed, mark task as done */
        if ( now >= next_switch_time)
            task_done[dom_priv->parent] = true;

        while ( ( (now >= next_switch_time)
               || (!dom_priv->healthy)
               || (task_done[dom_priv->parent]) )
               && (sched_index < sched_priv->num_schedule_entries) )
        {
            /* time to switch to the next domain in this major frame */
            sched_index++;
            if (sched_index < sched_priv->num_schedule_entries) {
                dom_priv = DOM_PRIV(sched_priv->schedule[sched_index].vc->domain);
                if (!task_done[dom_priv->parent])
                    next_switch_time += sched_priv->schedule[sched_index].runtime;
            }
        }
    }

    /*
     * If we exhausted the domains in the schedule and still have time left
     * in the major frame then switch next at the next major frame.
     */
    if ( sched_index >= sched_priv->num_schedule_entries )
        next_switch_time = sched_priv->next_major_frame;
```

```
    /*
     * If there are more domains to run in the current major frame, set
     * new_task equal to the address of next domain's VCPU structure.
     * Otherwise, set new_task equal to the address of the idle task's VCPU
     * structure.
     */
    new_task = (sched_index < sched_priv->num_schedule_entries)
       ? sched_priv->schedule[sched_index].vc
       : IDLETASK(cpu);

    /* Check to see if the new task can be run (awake & runnable). */
    if ( !((new_task != NULL)
          && (AVCPU(new_task) != NULL)
          && AVCPU(new_task)->awake
          && vcpu_runnable(new_task)) )
       new_task = IDLETASK(cpu);
    BUG_ON(new_task == NULL);

    /*
     * Check to make sure we did not miss a major frame.
     * This is a good test for robust partitioning.
     */
    BUG_ON(now >= sched_priv->next_major_frame);

    spin_unlock_irqrestore(&sched_priv->lock, flags);

    /* Tasklet work (which runs in idle VCPU context) overrides all else. */
    if ( tasklet_work_scheduled )
       new_task = IDLETASK(cpu);

    /* Running this task would result in a migration */
    if ( !is_idle_vcpu(new_task)
        && (new_task->processor != cpu) )
       new_task = IDLETASK(cpu);

    /*
     * Return the amount of time the next domain has to run and the address
     * of the selected task's VCPU structure.
     */
    ret.time = next_switch_time - now;
    ret.task = new_task;
    ret.migrated = 0;

    BUG_ON(ret.time <= 0);

    return ret;
}
```