Jurnal Sistem Informasi ISSN P : 2598-599X; E: 2599-0330

ANALISA PENGARUH UKURAN BUFFER UNTUK SISTEM DUPLIKASI FILE BERSKALA BESAR PADA TEMPORARY FILE SYSTEM

Himmatur Rijal[1], Balqis Yafis[2]
Sistem Informasi Universitas Malikussaleh[1]
EECS IGP National Yang Ming Chiao Tung University[2]
Jl. Cot Tgk Nie-Reulet, Aceh Utara, 141 Indonesia
email: himmatur.rijal@unimal.ac.id[1], balqisyafis.ee10@nycu.edu.tw[2]

Abstrak

Dalam penelitian ini, penulis mengidentifikasi pengaruh ukuran *buffer* dalam proses duplikasi file dengan ukuran file 1 gigabyte (GB) dan menggunakan sistem file *tmpfs*. Baseline code menggunakan operasi tingkat-rendah (*low-level operation*) seperti open(), creat(), read(), dan write(). Penelitian ini bertujuan untuk mengoptimasi proses duplikasi file menggunakan tiga metode. Pada tahap awal, penulis mengidenfikasi beberapa isu yang terdapat pada operasi read/write pada sistem file linux. Terdapat tiga metode yang akan diinvestgasi dalam penelitian ini, pertama, meningkatkan ukuran buffer, kedua membandingkan fungsi creat() dan open(), ketiga menganalisa fungsi O_Direct() pada tmpfs. Untuk duplikasi file, ukuran buffer menjadi hambatan kinerja pada proses sistem file ini. Karena jumlah read() tergantung pada ukuran *buffer*, sehingga operasi baca-tulis terus beralih antara mode pengguna dan mode kernel. Adapun analisa sistem ini menggunakan Sistem Operasi Linux.

Kata Kunci—Sistem Operasi, Linux, Sistem file, tmpfs.

Abstract

In this study, the authors identified the effect of buffer size on duplicating files with a 1 gigabyte (GB) file size using a ram-based file system, namely tmpfs. Baseline code uses low-level operations such as open(), create(), read(), and write(). This study aims to optimize the file duplication process using three methods. In the early stages, the author identifies several issues in the Linux file system's read/write operations. Three approaches will be invested in this research: increasing the buffer size and comparing the create() and open() functions. Thirdly analyze the O_Direct() role on tmpfs. For file duplication, the buffer size becomes a performance bottleneck in these file system processes. Because the number of read() depends on the size of the buffer; thus the read-write operations keep switching between user mode and kernel mode. The system analysis uses the Linux Operating System.

Keywords—Operating System, Linux, File System, tmpfs.

1. PENDAHULUAN

Banyak pengguna komputer melakukan transfer data yang berukuran sangat besar dimana melibatkan proses membaca data dari disk dan menulis kembali data yang sama persis ke soket respons. Sistem file atau sistem berkas merupakan struktur

logika yang digunakan untuk mengendalikan akses terhadap data yang ada pada disk. Dengan kata lain, sistem file merupakan database khusus untuk penyimpanan, pengelolaan, manipulasi dan pengambilan data, agar mudah ditemukan dan diakses dan dapat juga dilihat melalui sistem monitoring RFID (Ula, et all.,2021) dan memerlukan data untuk file akses (Rahma, et.,all., 2021)

Hubungan antara sistem operasi dengan sistem file adalah sistem file (file system) merupakan interface yang menghubungkan sistem operasi dengan disk (Marshall, 1984). Ketika program menginginkan pembacaan dari hard disk atau media penyimpanan lainnya, sistem operasi akan meminta sistem file untuk mencari lokasi dari file yang diinginkan. Setelah file ditemukan, sistem file akan membuka dan membaca file tersebut, kemudian mengirimkan informasinya kepada sistem operasi dan akhirnya bisa dibaca oleh pengguna.

Aktivitas transfer data ini tergolong tidak efisien karena kernel membaca data dari disk dan mendorongnya melintasi batas user-kernel ke aplikasi, dan kemudian aplikasi mendorongnya kembali melintasi batas user-kernel untuk ditulis ke soket. Akibatnya, aplikasi berfungsi sebagai perantara yang tidak efisien yang mendapatkan data dari file disk ke soket. Setiap kali data melintasi batas user dan kernel, data harus disalin, proses transfer data ini mengakibatkan terjadinya overhead. Dimana akan menghabiskan siklus CPU dan bandwidth memori.

2. TINJAUAN PUSTAKA

2.1 Sistem File Linux

Sistem operasi Linux mendukung banyak sistem file yang berbeda, tapi pilihan yang umum digunakan adalah keluarga Ext* (Ext2, Ext3 dan Ext4) (Gufron, 2016). Berikut sistem file yang umumnya digunakan pada sistem operasi Linux:

2.1.1 Ext2 (2nd Extended)

Sistem file Ext2 menyimpan data secara hirarki standar yang banyak digunakan oleh sistem operasi. Data tersimpan di dalam file, file tersimpan di dalam direktori. Sebuah direktori bisa mencakup file dan direktori lagi di dalamnya yang disebut sub direktori. Ext2 mendefinisikan topologi sistem file dengan memberikan arti bahwa setiap file pada sistem diasosiasikan dengan struktur data *inode*. Sebuah *inode* menunjukkan blok mana dalam suatu *file* tentang hak akses setiap *file*, waktu modifikasi file, dan tipe file. Setiap file dalam sistem file Ext2 terdiri dari inode tunggal dan setiap inode mempunyai nomor identifikasi yang unik. Inode-inode file sistem disimpan dalam tabel inode.

2.1.1 Third Extended File System (Ext3)

Sistem file Ext3 adalah peningkatan dari sistem file Ext2. EXT3 merupakan suatu journalled filesystem. Journalled filesystem didesain untuk membantu melindungi data yang ada di dalamnya. Dengan adanya journalled filesystem, maka kita tidak perlu lagi untuk melakukan pengecekan ke-konsistensian data, yang akan memakan waktu sangat lama bagi harddisk yang berkapasitas besar. EXT3 adalah suatu filesystem yang dikembangkan untuk digunakan pada sistem operasi Linux. EXT3 merupakan hasil perbaikan dari EXT2 ke dalam bentuk EXT2 yang lebih baik dengan menambahkan berbagai macam keunggulan diantaranya: availability, integritas data, Ext3 menjamin adanya integritas data setelah terjadi kerusakan atau unclean shut down. Ext3 memungkinkan kita memilih jenis dan tipe proteksi dari data. Dalam hal kecepatan, Ext3 mempunyai throughput yang lebih besar daripada Ext2 karena Ext3 memaksimalkan pergerakan head hard disk.

2.1.2 Fourth Extended File System (Ext4)

Ext4 dirilis secara utuh dan stabil berawal dari kernel 2.6.28. Keuntungan yang bisa didapat dengan mengupgrade filesystem ke ext4 dibanding ext3 adalah mempunyai pengalamatan 48-bit block yang artinya dia akan mempunyai 1EB = 1,048,576 TB ukuran maksimum filesystem dengan 16 TB untuk maksimum file size nya, fast fsck, journal checksumming, dan defragmentation support.

2.2 Temporary File System (tmpfs)

Tmpfs (Temporary File System) adalah paradigma penyimpanan file sementara yang diterapkan di banyak sistem operasi mirip Unix. Ini dimaksudkan untuk tampil sebagai sistem file yang terpasang (mounted file system), data disimpan dalam memori yang mudah menguap (*volatile*). Konstruksi tmpfs serupa dengan disk RAM, yang muncul sebagai drive disk virtual untuk menjalankan sistem file di dalam disk.

Tmpfs didukung oleh kernel Linux mulai dari versi 2.4. Linux tmpfs (sebelumnya dikenal sebagai shmfs) didasarkan pada kode ramfs yang digunakan selama bootup dan juga menggunakan cache halaman, tetapi tidak seperti ramfs, Linux mendukung penggantian halaman yang jarang digunakan untuk menukar ruang, serta ukuran sistem file dan batas inode untuk mencegahnya. situasi memori (default masing-masing setengah dari RAM fisik dan setengah dari jumlah halaman RAM) (Michael,2010).

3. METODELOGI PENELITIAN

3.1 Kerangka Kerja Penelitian

Pada penelitian ini kerangka kerja utamanya ditampilkan pada gambar 1.

Gambar 1. Kerangka Kerja Penelitian

Adapun tahapan-tahapan dalam penelitian ini berdasarkan gambar 1 adalah:

1. Observasi Masalah

Dalam penelitian ini, penulis melakukan optimasi aktivitas salin data dengan ukuran 1GB dan menggunakan sistem file tmpfs. Aktivitas salin data berskala besar pada linux menggunakan metode ukuran buffer tergolong tidak efisien karena jumlah pembacaan data tergantung pada ukuran buffer. Sehingga, setiap kali data melintasi batas user dan kernel, data harus disalin, proses transfer data ini mengakibatkan terjadinya overhead. Dimana akan menghabiskan siklus CPU dan bandwidth memori.

2. Perumusan Masalah

Bagaimana mengurangi jumlah pembacaan dan penulisan data sehingga mengurangi lalu lintas di batas user dan kernel.

3. Menganalisa Kebutuhan

Memerlukan kapasitas RAM minimal 3GB dan sistem operasi linux.

4. Perancangan Sistem

Memodifikasi kode yang meminimalisasi jumlah read() dan write().

5. Pengimplementasian

Untuk pengimplementasian sistem dilakukan pada command prompt sistem operasi Linux dengan spesifikasi hardware sebagai berikut:

- Linux virtual machine dengan 8GB RAM
- Intel core i7-7700 Ubuntu 22.04 1 Core (3.6 GHz – 4.2 GHz)

6. Melakukan Evaluasi Hasil

Hasil implementasi kemudian akan dievaluasi menurut waktu komputasi.

3.2 Analisis Baseline

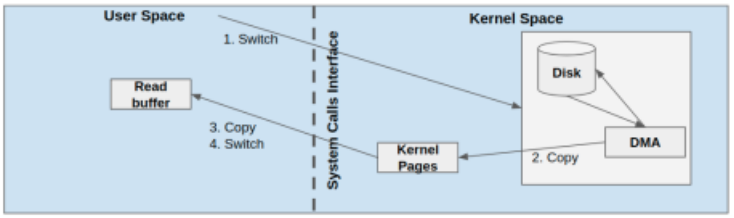
Dalam penelitian ini, penulis melakukan duplikasi file berukuran 1GB untuk mempelajari bagaimana proses *copy* file mempengaruhi *overhead* di ruang *user* dan ruang *kernel*. Menggunakan primitif kernel dalam program ini memungkinkan kita untuk langsung masuk ke kernel mode dari mode pengguna. Seperti yang bisa dilihat dari metode baseline pada gambar 1 berikut, dimana untuk masuk ke mode kernel secara langsung menggunakan system call open() lebih efisien daripada fopen().

Pada dasarnya, jika kita ingin menyalin program, kita perlu membuka file terlebih dahulu sebelum membacanya file, kemudian kita menulis isi file yang telah dibaca ke file tujuan. Oleh karena itu, terdapt empat *system call* penting yang terlibat dalam program ini, yaitu open(), read(), write() dan close(). Sedangkan pada kode baseline menggunakan lima *system call* yaitu: open(), creat(), read(), write() dan close().

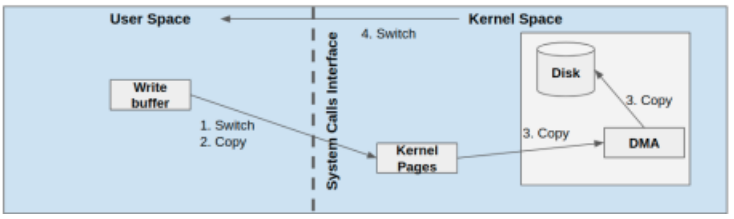
Ukuran buffer (N) pada kode baseline menjadi hambatan kinerja pada proses sistem file ini. Karena jumlah read() tergantung pada ukuran buffer, sehingga operasi baca-tulis terus beralih antara mode pengguna dan mode kernel. Untuk meminimalkan overhead, kita perlu menambah ukuran buffer agar read count() dapat berkurang, sehingga meminimalisasi perpindahan dari mode pengguna ke mode kernel.

```
while(1){
    readn = read(fd_in, buf, BUF_SIZE);
    if(readn <= 0){
        break;
    }
    while(readn){
        size_t writen = write(fd_out, buf, readn);
        if(writen == -1){
            perror("write() failed");
            return -1;
        }
        readn -= writen;
    }
}
```

Gambar 2 Kode baseline



Gambar 3 Operasi read dan overhead



Gambar 4 Operasi wrtie dan overhead

Ada tiga metode yang akan kita bahas, pertama meningkatkan ukuran buffer, kedua mengganti fungsi creat () dengan fungsi open () untuk menangani operasi tulis, dan akhirnya menggunakan O_DIRECT untuk membaca dan menulis. Secara keseluruhan, modifikasi yang dibuat berdasarkan baseline dapat dilihat pada Tabel 1.

Tabel 1 Metode Perancangan

	Baseline	Method 1		Method 2		Method 3	
Buffer size	1024 Bytes read(_, 1024)	4096	8192	4096	8192	4096	8192
Read	open (_, O_RDONLY)	open (_, O_RDONLY)		open(, O_RDONLY)		open (_, O_RDONLY | O_DIRECT)	
Write	creat(_, 0644);	creat(_, 0644);		open(_, O_WRONLY| O_CREAT , 0644)		open (_, O_WRONLY | O_DIRECT,0644)	

3.2.1 Metode I

Menurut kode *baseline*, ukuran *buffer* adalah 1024 byte, jadi jumlah read() menjadi sangat tinggi ketika menggunakan ukuran *buffer* yang kecil (N). Oleh karena itu, posisi kursor akan bergeser 1024 setiap kali kita membaca 1024 byte. Dengan ukuran file 1GB=10243 itu berarti hitungan read() akan menjadi 10242 kali, dimana sangat memakan waktu komputasi atau time overhead. Jadi, untuk meningkatkan program penyalinan file, kita harus mengurangi jumlah read() dengan meningkatkan ukuran buffer. Dalam metode ini, penulis meningkatkan ukuran buffer (N) dari 1024 menjadi

4096 dan 8192 byte untuk mengurangi jumlahnya panggilan sistem (yaitu read() count), yang berarti bahwa N byte data dibaca sekaligus, dan kemudian ditulis secara berurutan.

Karena ukuran buffer yang paling banyak digunakan adalah 4K atau 8K secara default, maka kami melompat ke mendefinisikan nomor itu. Akibatnya, menambah ukuran buffer dapat membuat program disalin lebih cepat. Pertanyaan utamanya adalah seberapa besar buffer dengan metode baseline? Jadi, saat proses inisialisasi, ukuran buffer yang lebih besar sampai meraih titik jenuh. Akibatnya, fault segmentation error terjadi saat ukuran buffer 8MB.

3.2.2 Metode II

Untuk menyalin sebuah program, pertama-tama kita harus membaca file sumbernya. Sebelum membaca sumbernya file, kita harus membukanya terlebih dahulu. Karena open() memiliki fungsi yang sama dengan creat(), Penulis menginvestigasi perbedaan dari kedua fungsi tersebut. Dalam sebuat literature dijelaskan bahwa creat() sudah usang dalam versi modern POSIX spesifikasi [3]. Selanjutnya, disebutkan di halaman manual Linux bahwa fungsi creat() redundan [4]. Oleh karena itu, pada langkah ini penulis akan menginvestigasi seberapa redundan fungsi creat() tersebut. Kemudian, penulis akan membandingkannya dengan fungsi open() menggunakan flag O_CREAT dalam metode yang diusulkan kedua.

3.2.3 Metode III

O_DIRECT adalah flag dalam system call open(), yang memungkinkan transfer data sinkron. Menggunakan flag O_Direct sangat bermanfaat jika kita ingin menyalin file besar sekali saja tanpa polusi cache. Oleh karena itu, pada bagian ini kami mencoba menangani operasi read() dengan O_DIRECT. Karena tmpfs hanya mengizinkan file berada di RAM dan bukan pada penyimpanan/drive kami akan menguji metode ini lebih lanjut.

3.3 Hasil dan Pembahasan

Ada tiga metode yang akan diuji dalam penelitian ini, pertama meningkatkan ukuran buffer, kedua mengganti fungsi creat () dengan fungsi open () untuk menangani operasi write, dan menggunakan flag O_DIRECT untuk proses membaca dan menulis.

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#define BUF_SIZE 8192
//#define BUF_SIZE 4096
//#define BUF_SIZE 1048576
int main(int argc, char *argv[]){
if(argc != 3){
printf("Usage: %s <source> <destination>\n", argv[0]);
return -1;
}
int fd_in = open(argv[1], O_RDONLY);
if(fd_in == -1){
perror("readwrite: open");
return -1;
}
int fd_out = open(argv[2], O_WRONLY| O_CREAT,0644);
if(fd_out == -1){
perror("readwrite: creat");
return -1;
}

char buf[BUF_SIZE];
size_t readn;
while(1){
readn = read(fd_in, buf, BUF_SIZE);
if(readn <= 0){
break;
}
while(readn){
size_t writen = write(fd_out, buf, readn);
if(writen == -1){
perror("readwrite: write");
return -1;
}
readn -= writen;
}
}
if(readn == -1){
perror("readwrite: read");
return -1;
}
close(fd_in);
close(fd_out);
return 0;
}
```

a) Metode I

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#define BUF_SIZE 8192
//#define BUF_SIZE 4096
int main(int argc, char *argv[]){
if(argc != 3){
printf("Usage: %s <source> <destination>\n", argv[0]);
return -1;
}
int fd_in = open(argv[1], O_RDONLY);
if(fd_in == -1){
perror("readwrite: open");
return -1;
}
int fd_out = open(argv[2], O_WRONLY| O_CREAT,0644);
if(fd_out == -1){
perror("readwrite: creat");
return -1;
}

char buf[BUF_SIZE];
size_t readn;
while(1){
readn = read(fd_in, buf, BUF_SIZE);
if(readn <= 0){
break;
}
while(readn){
size_t writen = write(fd_out, buf, readn);
if(writen == -1){
perror("readwrite: write");
return -1;
}
readn -= writen;
}
}
if(readn == -1){
perror("readwrite: read");
return -1;
}
close(fd_in);
close(fd_out);
return 0;
}
```

b) Metode II

Gambar 5 Algoritma Metode I dan Metode 2

4. HASIL DAN PEMBAHASAN

4.1 Hasil Analisis Implementasi Metode I

Tahap awal:

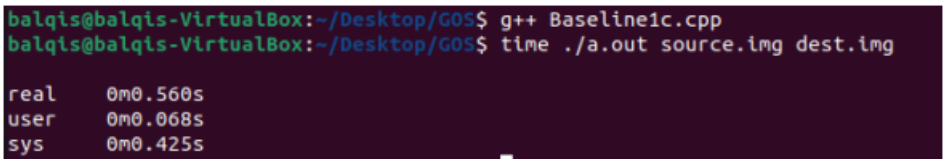
1. Mount tmpfs to the directory . → $ Sudo mount tmpfs tmpfs <mountpoint> -o size=3G
2. Allocate 1GB to copy later. → $ fallocate -l 1G source.img
3. Run the program. → $ g++ method1.cpp

Tabel 2 Hasil Analisis Metode I: meningkatkan ukuran buffer

	Baseline	Proposed Method 1			
	Creat()	Creat() – Increase Buffer			
BUF_SIZE	**1024**	**4096**	**8192**	**1MB**	**8MB**
Time (s)	2.283s	0.785s	0.560s	0.359s	Fault segmentation (core dump)
Time	100%	34%	25%	16%	-

```
balqis@balqis-VirtualBox:~/Desktop/GOS$ g++ Baseline1b.cpp
balqis@balqis-VirtualBox:~/Desktop/GOS$ time ./a.out source.img dest.img

real    0m0.785s
user    0m0.118s
sys     0m0.584s
```



Gambar 1 Ukuran Buffer 8192 Bytes

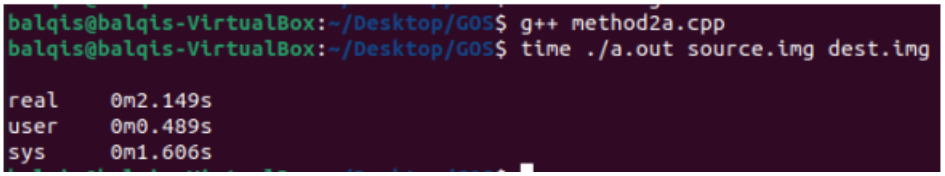
Gambar 2 Ukuran Buffer 1 MB

4.2 Hasil Analisis Implementasi Metode II

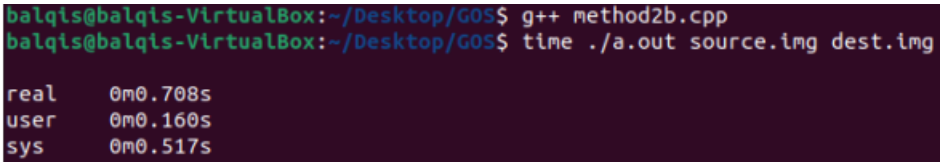
Tahap awal:

Tabel 3 Hasil Analisis Metode III

	Baseline	Proposed Method 1		Proposed Method 2			
	Creat()	Creat() – Increase Buffer		Open (... , O_CREAT)			
BUF_SIZE	1024	4096	8192	1024	4096	8192	1M
Time (s)	2.283s	0.785s	0.560s	2.149s	0.708s	0.544s	0.334s
Time	100%	34%	25%	94%	31%	23.8%	14.6%

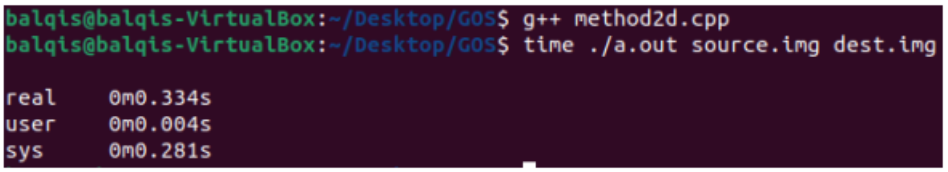


Gambar 3 Ukuran buffer 1024 Bytes



Gambar 4 Ukuran buffer 4096 Bytes

Gambar 5 Ukuran buffer 8192 Bytes



Gambar 6 Ukuran buffer 1 MB

Dengan membandingkan hasil di atas, kita dapat melihat perbedaan antara creat() dan open(), dan hasilnya menunjukkan adanya penurunan waktu komputasi walaupun tidak terlalu signifikan. Karena itu kita dapat menyimpulkan bahwa open() dan creat() itu setara meskipun creat() dicatat sebagai redundan di Linux halaman manual [1]. Oleh karena itu, kami menyimpulkan bahwa ukuran buffer adalah hambatan utama dalam hal ini.

5. KESIMPULAN

Penelitian ini menyimpulkan bahwa meningkatkan ukuran buffer adalah metode terbaik yang kami miliki percobaan untuk mengurangi overhead saat menduplikasi file besar di tmpfs. Karena tmpfs adalah penyimpanan sistem file sementara di memori RAM. Meskipun meningkatkan ukuran buffer dapat membuatnya lebih cepat tetapi memiliki keterbatasan. Pertama, tergantung pada ukuran memori, jika ukuran memori kecil dan sibuk, meningkatkan ukuran buffer dapat menyebabkan sistem mati, lambat atau mengalami saturasi. Kedua adalah kesalahan segmentasi yang terjadi ketika ukuran buffer mencapai 8 MB, hal ini terjadi karena kapasitas cache hardware yang ada di Core i7-7700 adalah 8MB.

DAFTAR PUSTAKA

Fitria, R., Yulisda, D., & Ula, M. (2021). Data Mining Classification Algorithms For Diabetes Dataset Using Weka Tool. Sisfo: Jurnal Ilmiah Sistem Informasi, 5(2).

Gufron. (2016). Perbaikan Perangkat Komputer oleh Bung Hatta University Press.

Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. (1984). A fast file system for UNIX. ACM Trans. Comput. Syst. 2, 3 (August 1984), 181-19

Michael Kerrisk. (2010). The Linux Programming Interface: A Linux and UNIX System Programming Handbook. 1st. USA: No Starch Press, 2010. isbn: 1593272200

Mark Mitchell and Alex Samuel. (2001). Advanced Linux Programming. USA: New Riders Publishing, 2001. isbn: 0735710430.

Ula, M., Pratama, A., Asbar, Y., Fuadi, W., Fajri, R., & Hardi, R. (2021, April). A New Model of The Student Attendance Monitoring System Using RFID Technology. In Journal of Physics: Conference Series (Vol. 1807, No. 1, p. 012026). IOP Publishing.

Ula, M., Darnila, E., Siagian, P., Siagian, L., & Sinambela, M. (2018, September). Machine learning on waveform spectral analysis of nuclear explosion from broadband seismic station in Indonesia. In IOP Conference Series: Materials Science and Engineering (Vol. 420, No. 1, p. 012047). IOP Publishing.

Retno, S., Hasdyna, N., Mutasar, M., & Dinata, R. K. (2020). Algoritma Honey Encryption dalam Sistem Pendataan Sertifikat Tanah dan Bangunan di Universitas Malikussaleh. INFORMAL: Informatics Journal, 5(3), 87-95.

Robbins, Daniel. (2001). "Common threads: Advanced filesystem implementor's guide, Part 3. IBM Developers Works, diakses 2022-06-06.