

## **Analisa Pengaruh Ukuran Buffer Untuk Sistem Duplikasi File Berskala Besar Pada Temporary File System**

Himmatur Rijal<sup>1</sup>, Balqis Yafis<sup>2</sup>

Sistem Informasi Universitas Malikussaleh<sup>1</sup>

EECS IGP National Yang Ming Chiao Tung University<sup>2</sup>

Jl. Cot Tgk Nie-Reulet, Aceh Utara, 141 Indonesia

email: himmatur.rijal@unimal.ac.id<sup>1</sup>, balqisyafis.ee10@nycu.edu.tw<sup>2</sup>

### **Abstrak**

Dalam penelitian 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 mengidentifikasi beberapa isu yang terdapat pada operasi `read/write` pada sistem file linux. Terdapat tiga metode yang akan diinvestigasi 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 socket 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 al., 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 socket. Akibatnya, aplikasi berfungsi sebagai perantara yang tidak efisien yang mendapatkan data dari file disk ke socket. 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) (Gufon, 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.. 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.

### 2.2 Temporary File System (tmpfs)

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

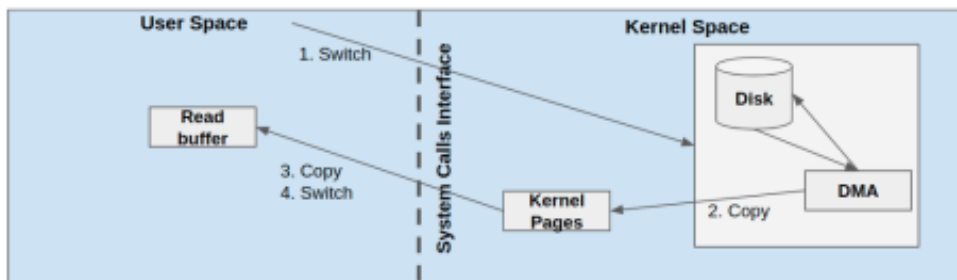
Dalam penelitian ini, penulis melakukan duplikasi file berukuran 1GB untuk mempelajari bagaimana proses *copy* file mempengaruhi *overhead* di ruang *user* dan ruang *kernel*. Menggunakan primitif kernel dalam program ini memungkinkan kita untuk langsung masuk ke kernel mode dari mode pengguna. Seperti 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, terdapat 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()`.

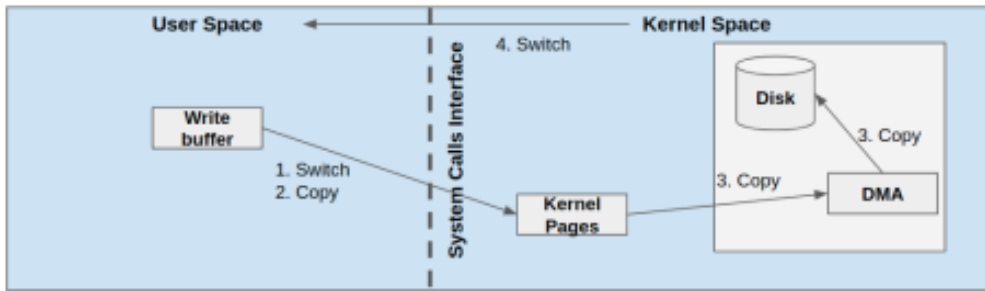
Ukuran buffer (N) pada kode baseline menjadi hambatan kinerja pada proses sistem file ini. Karena jumlah read() tergantung pada ukuran buffer, sehingga operasi baca-tulis terus beralih antara mode pengguna dan mode kernel. Untuk meminimalkan overhead, kita perlu menambah ukuran buffer agar read count() 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	
<b>Buffer size</b>	1024 Bytes <code>read(_, 1024)</code>	4096	8192	4096	8192	4096	8192
<b>Read</b>	<code>open(_, O_RDONLY)</code>	<code>open(_, O_RDONLY)</code>		<code>open(_, O_RDONLY)</code>		<code>open(_, O_RDONLY   O_DIRECT)</code>	
<b>Write</b>	<code>creat(_, 0644);</code>	<code>creat(_, 0644);</code>		<code>open(_, O_WRONLY   O_CREAT, 0644)</code>		<code>open(_, O_WRONLY   O_DIRECT, 0644)</code>	

### 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 inialisasi, 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 sebuah 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.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #define BUF_SIZE 8192
6 // #define BUF_SIZE 4096
7 // #define BUF_SIZE 1048576
8 int main(int argc, char *argv[]){
9     if(argc != 3){
10        printf("Usage: %s <source> <destination>\n", argv[0]);
11        return -1;
12    }
13    int fd_in = open(argv[1], O_RDONLY);
14    if(fd_in == -1){
15        perror("readwrite: open");
16        return -1;
17    }
18    int fd_out = open(argv[2], O_WRONLY| O_CREAT, 0644);
19    if(fd_out == -1){
20        perror("readwrite: creat");
21        return -1;
22    }
23
24    char buf[BUF_SIZE];
25    size_t readn;
26    while(1){
27        readn = read(fd_in, buf, BUF_SIZE);
28        if(readn <= 0){
29            break;
30        }
31        while(readn){
32            size_t writen = write(fd_out, buf, readn);
33            if(writen == -1){
34                perror("readwrite: write");
35                return -1;
36            }
37            readn -= writen;
38        }
39        if(readn == -1){
40            perror("readwrite: read");
41            return -1;
42        }
43        close(fd_in);
44        close(fd_out);
45        return 0;
46    }
47 }
48
                
```

**a) Metode I**

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5 #define BUF_SIZE 8192
6 // #define BUF_SIZE 4096
7 int main(int argc, char *argv[]){
8     if(argc != 3){
9        printf("Usage: %s <source> <destination>\n", argv[0]);
10        return -1;
11    }
12    int fd_in = open(argv[1], O_RDONLY);
13    if(fd_in == -1){
14        perror("readwrite: open");
15        return -1;
16    }
17    int fd_out = open(argv[2], O_WRONLY| O_CREAT, 0644);
18    if(fd_out == -1){
19        perror("readwrite: creat");
20        return -1;
21    }
22
23    char buf[BUF_SIZE];
24    size_t readn;
25    while(1){
26        readn = read(fd_in, buf, BUF_SIZE);
27        if(readn <= 0){
28            break;
29        }
30        while(readn){
31            size_t writen = write(fd_out, buf, readn);
32            if(writen == -1){
33                perror("readwrite: write");
34                return -1;
35            }
36            readn -= writen;
37        }
38        if(readn == -1){
39            perror("readwrite: read");
40            return -1;
41        }
42        close(fd_in);
43        close(fd_out);
44        return 0;
45    }
46 }
47 }
48
                
```

**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=3GB
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/G05$ g++ Baseline1b.cpp
balqis@balqis-VirtualBox:~/Desktop/G05$ time ./a.out source.img dest.img

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

balqis@balqis-VirtualBox:~/Desktop/G05$ g++ Baseline1c.cpp
balqis@balqis-VirtualBox:~/Desktop/G05$ time ./a.out source.img dest.img

real    0m0.560s
user    0m0.068s
sys     0m0.425s
    
```

Gambar 1 Ukuran Buffer 8192 Bytes

```

balqis@balqis-VirtualBox:~/Desktop/G05$ g++ Baseline1e.cpp
balqis@balqis-VirtualBox:~/Desktop/G05$ time ./a.out source.img dest.img

real    0m0.359s
user    0m0.000s
sys     0m0.290s
    
```

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%

```

balqis@balqis-VirtualBox:~/Desktop/G05$ g++ method2a.cpp
balqis@balqis-VirtualBox:~/Desktop/G05$ time ./a.out source.img dest.img

real    0m2.149s
user    0m0.489s
sys     0m1.606s
    
```

Gambar 3 Ukuran buffer 1024 Bytes

```

balqis@balqis-VirtualBox:~/Desktop/G05$ g++ method2b.cpp
balqis@balqis-VirtualBox:~/Desktop/G05$ time ./a.out source.img dest.img

real    0m0.708s
user    0m0.160s
sys     0m0.517s
    
```

Gambar 4 Ukuran buffer 4096 Bytes

```
balqis@balqis-VirtualBox:~/Desktop/G05$ g++ method2c.cpp
balqis@balqis-VirtualBox:~/Desktop/G05$ time ./a.out source.img dest.img

real    0m0.544s
user    0m0.083s
sys     0m0.408s
```

Gambar 5 Ukuran buffer 8192 Bytes

```
balqis@balqis-VirtualBox:~/Desktop/G05$ g++ method2d.cpp
balqis@balqis-VirtualBox:~/Desktop/G05$ time ./a.out source.img dest.img

real    0m0.334s
user    0m0.004s
sys     0m0.281s
```

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

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

- 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.

SISFO : Jurnal Ilmiah Sistem Informasi. Vol 6, No 2 (2022)