

Group 8

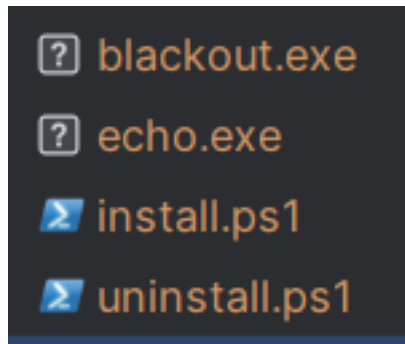
Emergency Backup

Programmazione di Sistema, Sep 2024

Emanuele Messina

Installing

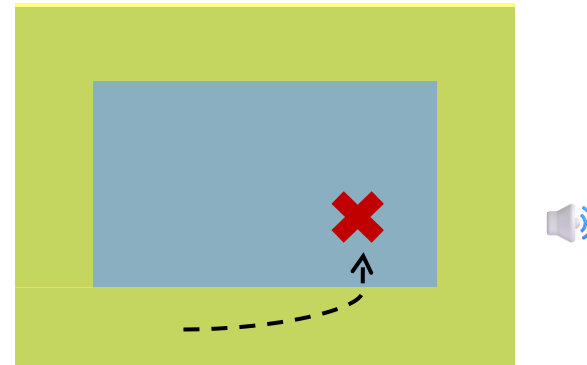
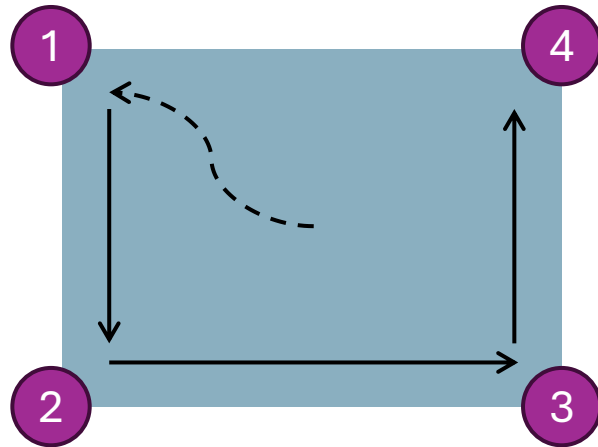
- Currently runs on Windows only (should be easy to port with minor adjustments)
- The program is portable by moving its release folder (terminate and uninstall before moving)
- Install/Uninstall scripts register/unregister a Scheduled Task to start the process at user logon



```
? blackout.exe  
? echo.exe  
> install.ps1  
> uninstall.ps1
```

Usage

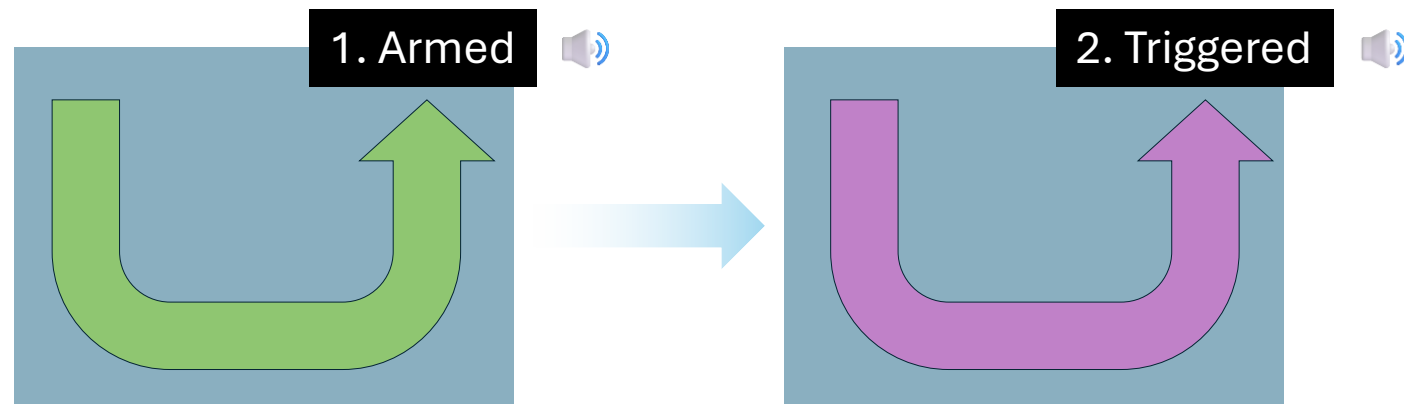
- Once the main process is running (blackout.exe) you can trace the following path with the mouse on the primary display



- When performing the path, the mouse pointer must be kept inside the safe region (thickness = $\frac{1}{6}$ of the screen width), otherwise the operation must start over

Usage

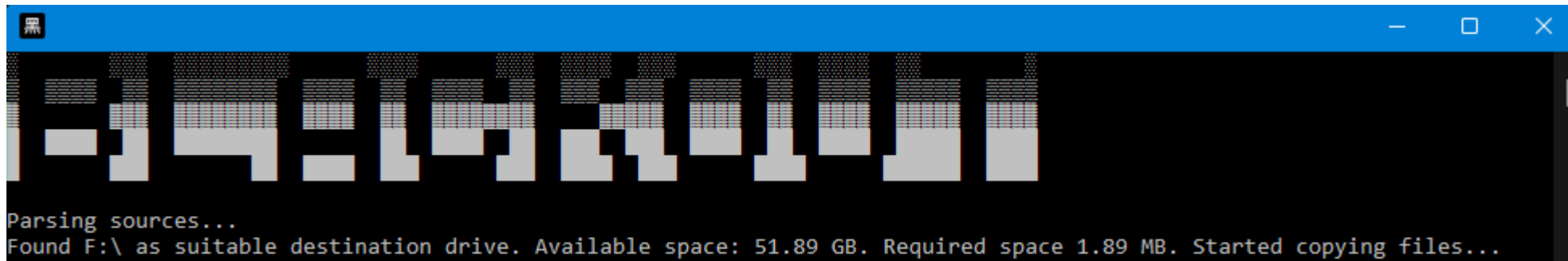
- To trigger the backup, the path must be completed two times in a row



- Once triggered, the backup logic starts and the mouse logic is disabled

Usage

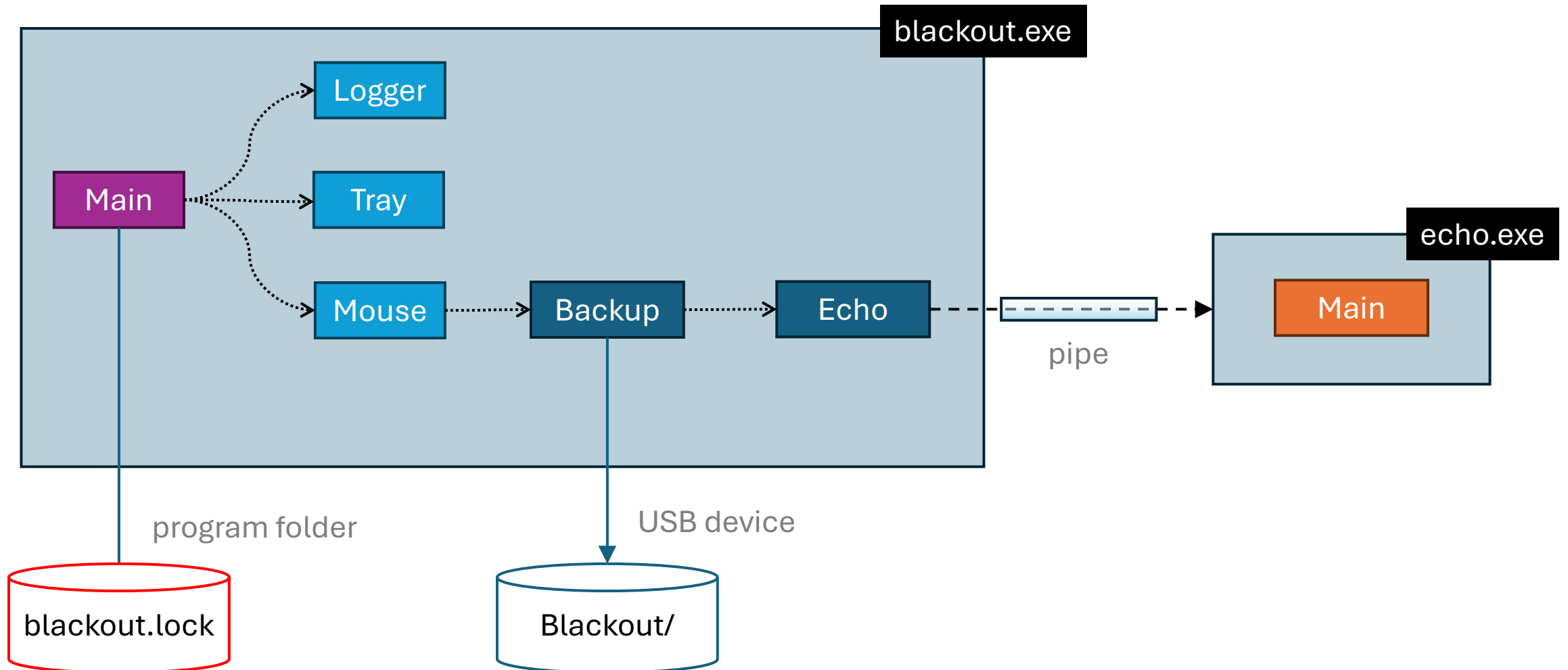
- The backup sources are specified as **glob** patterns in `sources.txt` in the program folder
- A console window will pop up, showing the backup status while in progress. If closed, it will reopen automatically.
- The files will start copying on the first suitable USB device found (one with enough free space to allocate all the files to backup)



Usage

- The files will be saved under `Blackout/<timestamp>` on the chosen device, following their original directory structure
- If a file fails, it will ignore it and try to complete the backup. Other errors cause the backup to fail.
- Either way, the backup logic terminates and the mouse logic restarts. The application quits by itself only if there are fatal errors that prevent the program from running.
- Each major event or state change is accompanied by a sound.
- The general application log is available as `blackout.log` in the program folder. The backup specific log is saved as `Blackout/<timestamp>.log`

Architecture



General considerations

- The application tries to minimize the active waiting time of the threads to keep the CPU usage as low as possible.
- Audio is played async by spawning tokio tasks.
- Resources (audio files, images) are embedded into the binary at compile time.

Logger

- Available to the entire application

```
pub static mut LOGGER: Logger = Logger { thread: None, tx: None };
```

- Abstraction macros

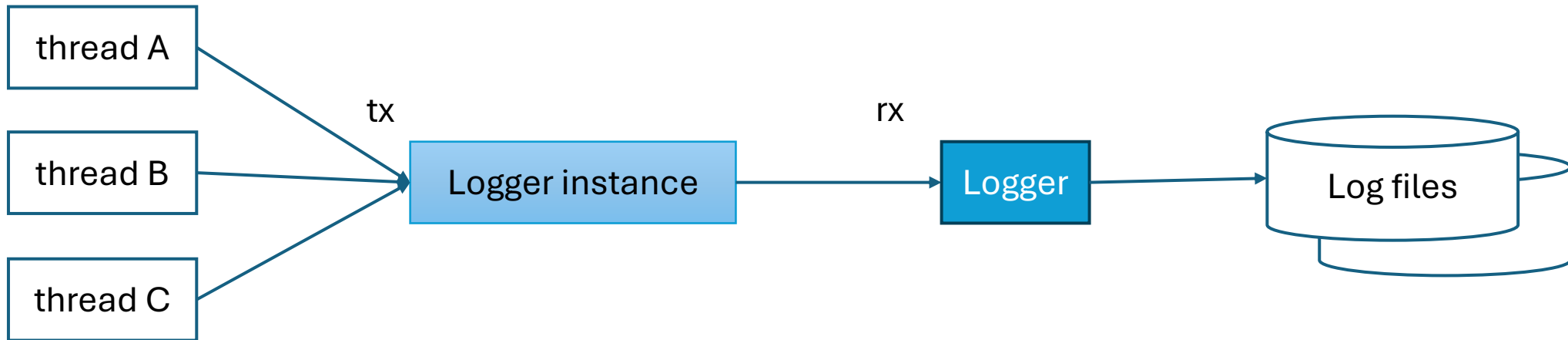
```
macro_rules! info {  
    ($from:expr, $body:expr) => {  
        unsafe {  
            crate::logger::LOGGER.log(crate::logger::LogLevel::Info, $from.to_string(), $body.to_string());  
        }  
    };  
}
```

- Example usage

```
info!("main", "Process started.");
```

Logger

- Threads send log messages to the logger thread via a channel: logging is asynchronous from the threads pov
- The logger loops on channel recv as usual
- It can route log messages to different log files



State

- There is a global state as well, available to all the threads
- Threads can subscribe to state changes, either synch or asynchronously: they can wait for a change with `crossbeam` or `tokio select!` macros, or read it directly
- Currently used only to signal the threads that the application is quitting, but opens up the possibility of more complex flows

```
lazy_static! {  
    pub static ref APP_STATE : Arc<ApplicationStateManager> = ApplicationStateManager::new();  
}  
  
pub struct ApplicationStateManager {  
    state: RwLock<ApplicationState>,  
    notify: Notify,  
    subscribers: RwLock<Vec<tokio::sync::mpsc::Sender<ApplicationState>>>,  
    subscribers_sync: RwLock<Vec<crossbeam::channel::Sender<ApplicationState>>>  
}
```

Tray

- Handles the tray icon UI in the taskbar
- Currently has a single Quit button to let the user terminate the application manually (only case in which the quit state is not because of a fatal error)
- Uses channels as a Rust abstraction over OS tray events

Main

- Enforces a single running instance of the executable via file locking
- Sets up the application and spawns Logger, Tray, and Mouse threads
- Logs the process CPU utilization every 2 minutes to the general log file
- Since it must be async to run tokio (for audio playing), it uses a tokio select macro to handle state changes and cpu log events (via tokio sleep)

Mouse

- Mouse position is available on demand at any time, so unfortunately it uses an old school do-sleep pattern
- The refresh rate is kept at 200ms not to overload the CPU
- Once the path detection goes to the triggered state, it spawns the backup thread and immediately joins, the loop is so halted until the backup finishes (successfully or not)

Backup

- Launches the Echo manager thread to which it sends the status messages to be displayed during the backup, this way UI messages are treated the same way as logs from the backup thread pov
- It plays a heartbeat sound periodically during copying, and an error sound if the entire backup fails.
- Logs the elapsed time during the full copy
- When it finishes (successfully or not), it drops the Echo tx after 5 seconds and joins the Echo thread (Echo knows to terminate on tx drop)

Echo

- `echo.exe` is a separate binary that prints back whatever is sent to its stdin
- It's spawned by the Echo thread with piped stdio
- It manually allocates a console window on which to print the messages for formatting purposes (by default we would watch someone typing without ever pressing Enter...)
- The Echo thread recvs the messages from the Backup thread and forwards them to the echo process pipe
- It appears impossible to customize the console window on Windows: the user can close it and thus terminate the echo process
- The Echo thread monitors the echo process state (via `child try wait`) and respawns it if not alive
- Finally, it kills the echo process permanently and returns when Backup drops its tx (as an exit condition)