# Automatyczna cenzura twarzy

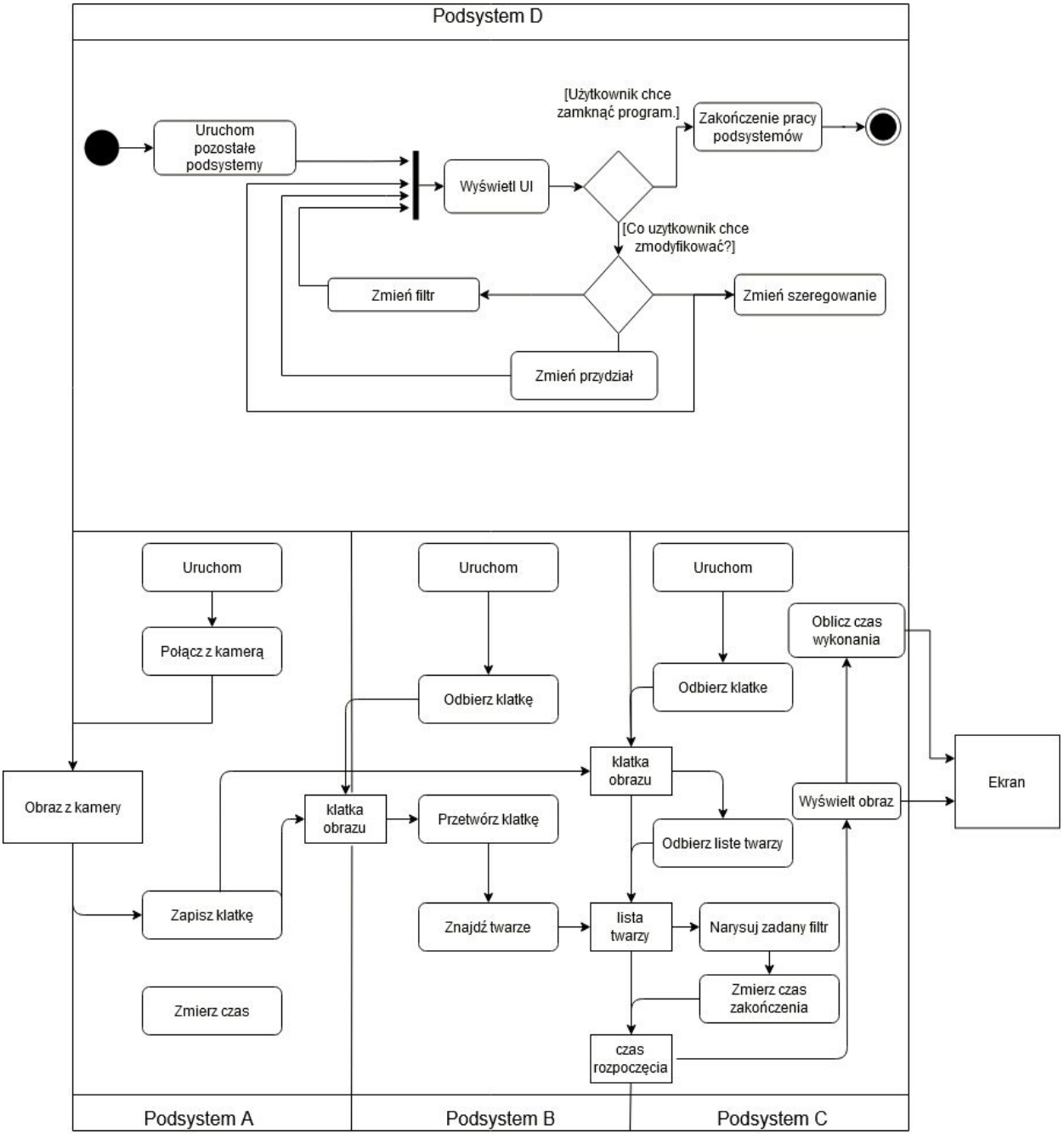
## Projekt realizowany w ramach przedmiotu
## Systemy Czasu Rzeczywistego

Paweł Budniak, Patryk Dobrowolski, Tymoteusz Perka

# Spis treści

# Wstęp

Celem realizowanego projektu jest stworzenie systemu, który odbiera obraz z kamery, w czasie rzeczywistym wykrywa w nim ludzkie twarze i nakłada na nie cenzurę. Użytkownik będzie miał możliwość wyboru formy cenzury (rozmazanie, zamalowanie) w trakcie działania systemu. Wynikowy obraz z nałożonymi już efektami będzie wyświetlany na ekranie komputera. Całość podzielona jest na 4 niezależnie, ale komunikujące się między sobą podsystemy. W ramach interfejsu użytkownika przewidziana jest także możliwość wpływania na sposób działania poszczególnych podsystemów. Projektowany system powinien odznaczać się przede wszystkim szybkością, tak, aby użytkownik mógł oglądać przetworzony obraz bez widocznych opóźnień.




Rysunek 1. Schemat systemu z podziałem na podsystemy

Na schemacie (Rysunek 1) widać przepływ danych przez zaprojektowany przez nas system. Najpierw obraz zarejestrowany przez kamerę jest odbierany przez podsystem A, który następnie rozsyła go do podsystemów B i C. Podsystem B ma za zadanie wykryć wszystkie ludzkie twarze znajdujące się w przetwarzanym obrazie oraz przekazanie informacji o ich położeniu do podsystemu C. On natomiast będzie nakładał pewien filtr uniemożliwiający rozpoznanie twarzy lub będzie je kompletnie zamazywał, w zależności od wybranej przez użytkownika opcji. Tak przetworzony obraz będzie wyświetlany na ekran. Podsystem D umożliwia użytkownikowi systemu kontrolę nad działaniem pozostałych, zasadniczych podsystemów poprzez zmianę parametrów ich działania. Podsystemy winne komunikować się za pomocą jak najszybszych kanałów komunikacji tak, aby zminimalizować opóźnienia niezwiązane

stricte z przetwarzaniem danych. Każdy z tych podsystemów powinien zostać maksymalnie zoptymalizowany pod kątem wykonywanego zadania, gdyż po to właśnie rozdzielamy nieco różniące się od siebie zadania pomiędzy 4 jednostki podwykonawcze.

## Wstęp teoretyczny do implementacji

Do realizacji wyżej przedstawionego systemu użyjemy programu komputerowego zaprojektowanego do uruchamiania wyłącznie na maszynie z systemem operacyjnym Linux. Ma to związek z koniecznością użycia narzędzi, które są udostępnianie tylko przez ten system. Pierwszym z nich jest możliwość tworzenia nowych procesów przez dany program za pomocą funkcji fork(). Funkcja ta kopiuje aktualnie wykonywany program do nowego procesu, który jest dzieckiem procesu, z poziomu którego wywołana została funkcja fork(). Rzadko kiedy potrzebujemy jednak dwóch dokładnie takich samych programów, więc często w parze z tą funkcją idzie funkcja execve(). Ta natomiast uruchamia w ramach wołającego procesu całkowicie nowy program, nie zostawiając żadnego śladu po poprzednio działającym. Użycie tych narzędzi pozwala z poziomu jednego głównego procesu (w naszym przypadku oznaczonego literą D
i odpowiadającego podsystemowi D) możemy stworzyć trzy pozostałe, zajmujące się zupełnie czymś innym niż on sam. Umożliwia to też kontrolę procesu D nad resztą systemu, gdyż funkcja fork() zwraca do wołającego numer identyfikacyjny (ang. *Process ID,* w skrócie *PID*) nowo utworzonego procesu , który później może zostać użyty do modyfikacji działania innych procesów. Każdy z podsystemów będzie reprezentowany przez jeden proces, więc na nasz program będą składały się cztery procesy. Zaletą takiego podejścia jest pełna niezależność poszczególnych komponentów systemu, można je niezależnie uruchamiać, zatrzymywać oraz edytować. W przypadku awarii jednego z naszych podsystemów (poza tym oznaczonym D, który jest krytyczny, bo zarządza pozostałymi) wystarczy jego ponowne uruchomienie, bez potrzeby resetowania pozostałych komponentów. Jedyną wadą takiego podejścia jest fakt, że pomimo, że nasze procesy składają się na jeden, działający spójnie system, to nie mają “wglądu” do zasobów innych uczestników tego systemu. Oznacza to, że proces odpowiadający za realizację podsystemu B nie jest w stanie zobaczyć obrazu zapisanego w pamięci operacyjnej przez proces A bez żadnej zewnętrznej pomocy. Jesteśmy w stanie sobie z tym poradzić za pomocą tak zwanej komunikacji międzyprocesowej (ang. *Interprocess communication*, w skrócie IPC). Systemy Uniksopodobne, a takim jest Linux, udostępniają wiele narzędzi umożliwiających taki zabieg. Najpopularniejszymi z nich są:

- sygnały - jest to asynchroniczna wiadomość bez żadnej treści poza typem sygnału (typy sygnałów są predefiniowane w systemie operacyjnym) przesyłana z jednego procesu do drugiego, która przerywa normalne

wykonywanie programu w drugim procesie i zmusza go do obsługi otrzymanej wiadomości. Rzadko używana forma ze względu na nieprzewidywalność zachowania oraz brak możliwości przesłania jakiejkolwiek treści.

- potoki - są to wirtualne pliki umożliwiające jednokierunkowy transfer dowolnych danych z jednego procesu do drugiego. Problemem tutaj jest fakt, że chcąc przesłać złożone dane musimy je ręcznie wyodrębnić z potoku w procesie drugim, gdyż potok przesyła po prostu ciąg bajtów.

- kolejki komunikatów - forma bardzo podobna do potoków z tą różnicą, że tutaj komunikacja jest nastawiona na komunikaty (czyli pewne zdefiniowane struktury danych), których nie trzeba już ręcznie wyodrębniać z kodu bajtowego. Minusem jest fakt, że dane są kopiowane dwukrotnie, raz przy nadawaniu oraz raz przy odbieraniu, co czyni tę formę komunikacji niewydajną przy dużych danych.

- pamięć współdzielona - jest to obszar pamięci operacyjnej, który może być współdzielony przez wiele procesów. Zachowuje się jak zwykła pamięć wewnętrzna procesu, jest więc bardzo szybka. Problemem tutaj jest synchronizacja dostępu do niej, gdyż jeżeli jeden proces będzie do niej coś zapisywał podczas gdy drugi będzie z niej odczytywał to może doprowadzić do nieprzewidywalnego zachowania programu.

Do takiej synchronizacji najczęściej używa się następujących narzędzi:

- mutex (ang. *mutual exclusion*, tłum. wzajemne wykluczanie) - zasób systemowy w postaci semafora, posiadający dwa stany: zamknięty oraz otwarty. Proces chcący dokonać zapisu lub odczytu do pamięci współdzielonej prosi o blokadę mutexu odpowiedzialnego za tą pamięć. Jeżeli mutex jest otwarty, to zostaje zablokowany, proces dokonuje zapisu i odczytu po czym otwiera go z powrotem. Natomiast jeżeli mutex jest zamknięty w wyniku działania innego procesu na pamięci współdzielonej, to proces chcący dostać blokadę zostanie uśpiony aż do momentu, kiedy proces drugi otworzy z powrotem mutex.

- rygle pętlowe - działają one na tej samej zasadzie co mutexy, tylko proces oczekujący na zwolnienie blokady nie jest usypiany, zamiast tego aktywnie oczekuje na zwolnienie. Ma to sens w przypadku, kiedy czas oczekiwania na zwolnienie zasobu jest na tyle krótki, że nie opłaca się tracić czasu na usypianie i wznawianie procesu.

- blokady zapis/odczyt - sposób działania także podobny do mutexu, różnicą jest fakt, że przy zastosowaniu tej blokady kilka procesów na raz może odczytywać zawartość pamięci współdzielonej.

Innym sposobem interakcji pomiędzy procesami będzie ustawianie sposobu szeregowania procesów (w szczególności ich priorytetów) za pomocą funkcji systemowej sched_setscheduler(). W systemie Linux mamy następujące sposoby szeregowania procesów:

- SCHED_OTHER - standardowa “karuzela” procesów, gdzie każdy proces dostaje taki sam kwant czasu procesora, po którym następuje przełączenie na następny proces. Po przejściu wszystkich procesów wracamy do pierwszego i tak w kółko.

- SCHED_BATCH - tryb szeregowania dedykowany dla procesów działających w tle, które potrzebują dużej szybkości działania oraz mają niewiele interakcji zewnętrznych.

- SCHED_IDLE - tryb stworzony dla procesów działających w tle, które mają niski priorytet i nie potrzebują dużo zasobów.

- SCHED_FIFO - kolejka priorytetowa procesów, gdzie proces o najwyższym priorytecie wykonuje się, dopóki się nie skończy lub nie zgłosi się proces o wyższym priorytecie, który wtedy przejmie czas procesora.

- SCHED_RR - tryb bliźniaczy do SCHED_OTHER z tym wyjątkiem, że kwant czasu przydzielony danemu procesowi zależy od priorytetu, który zostanie mu ustawiony.

Poza szeregowaniem możemy także wpływać na przydział poszczególnych procesów do rdzeni procesora. Jest to możliwe za pomocą funkcji systemowej sched_setaffinity(). Dzięki niej możemy danemu procesowi kazać działać np. tylko na rdzeniu nr 1 procesora, albo tylko na rdzeniach nr 2 i 3. Oczywiście ma to istotny wpływ na szybkość działania danego procesu.
W naszym systemie także będą zastosowane pewne algorytmy graficzne, głównie służące filtrowaniu. W przypadku projektowanego systemu interesują nas filtry rozmazujące obraz. Najpopularniejszymi z nich są:

- filtr uśredniający - z danego kwadratu o nieparzystej długości boku wyciąga średni kolor piksela (za pomocą średniej arytmetycznej wartości składowych R, G oraz B każdego piksela w kwadracie) po czym tak obliczoną wartość wstawia w miejsce środkowego piksela.
- filtr Gaussa - podobny do filtru uśredniającego, aczkolwiek używamy tutaj średniej ważonej, w której waga zależy od odległości danego piksela od środka kwadratu (im większa odległość, tym mniejsza waga).

## Platforma

Do implementacji naszego systemu użyjemy komputera działającego pod kontrolą systemu operacyjnego Linux Ubuntu 20.04 LTS ze zintegrowaną kamerą, języka C++ oraz następujących bibliotek:

- OpenCV - opensource'owa biblioteka znacznie ułatwiająca zaawansowaną obróbkę obrazu i przede wszystkim umożliwiająca rozpoznawanie obrazów.

- Boost.Interprocess - biblioteka upraszczająca komunikację międzyprocesową, czyniąca kod dużo bardziej czytelnym.

Oczywiście program zostanie zaprojektowany tak, aby współpracować z dowolną kamerą komputerową oraz konfiguracją komputera.

## Procesy

- Proces A - Odpowiada za odbiór obrazków (ramek) z kamery

  Klasa VideoCapture pozwala na odbiór obrazu z kamery. Po wywołaniu funkcji capture. open() proces będzie mógł zapisywać klatki obrazu i przekazywać do procesu B. Proces ten będzie także inicjować timer (zapisując po prostu aktualny timestamp w milisekundach), używając do tego biblioteki chrono.h, który następnie będzie przekazywany do kolejnych procesów w celu pomiaru wydajności całego procesu. Proces zapisuje klatki pojedynczo do pamięci współdzielonej, skąd mogą odczytać ją procesy B i C.

- Proces B - Odpowiada za rozpoznawanie twarzy i określenie, które piksele należą do niej.

  Do procesu przekazywane będzie klatka z kamery. Skorzystamy z wytrenowanego modelu dnn i pliku .prototxt, na podstawie których proces będzie rozpoznawać twarze. Następnie dana klatka zostanie podana do funkcji blobFromImage(). Na tak przygotowanych danych będzie można wygenerować macierz detekcji. Z macierzy będą wybierane obiekty, które są twarzami z prawdopodobieństwem większym niż 0.5. Znalezione twarze będą zapisane jako lista prostokątów vector<Rect>.

- Proces C - Odpowiada za nałożenie rozmazania lub zamalowania na piksele wskazane przez proces B

Na podstawie listy twarzy proces będzie modyfikować każdą klatkę. Dla każdego prostokąta będzie wydobywać jego współrzędne i rozmiary, a następnie nakładać zadany filtr. Dla filtru gaussowskiego istnieje funkcja GaussianBlur(). Innym sposobem blurowania twarzy może być podział prostokąta na bloki za pomocą funkcji linspace() i zamalowywanie bloków odcieniami wyliczonymi ze średniej początku i końca pojedynczego bloku. Zwykłe zamalowywanie prostokątów będzie wykonywane przez funkcję rectangle(frame ,(), -1), ostatni argument zapewnia wypełnienie środka figury. Proces ten będzie także generował aktualny, po wyświetleniu klatki, timestamp i porównywał go w celu uzyskania czasu przetwarzania danych przez system.

- Proces D - Zapewnia interfejs użytkownika, pozwalający na wybranie pomiędzy blurowaniem, a zamalowywaniem oraz zmiany parametrów wpływających na szeregowanie innych procesów

  Proces ten będzie działał w terminalu, gdzie wyświetlone zostanie menu użytkownika zawierające opcje: Modyfikuj przydział procesów do rdzeni, modyfikuj sposób szeregowania procesów oraz ich priorytet, zmień tryb cenzury, wyjście. Zanim wyświetli menu, utworzy i uruchomi procesy A, B, C za pomocą fork() + execve() i będzie trzymał ich identyfikatory w celu zarządzania ich szeregowaniem oraz ich zamknięciem z pomocą sygnału, który będzie obsługiwany przy użyciu zdefiniowanego handlera, aby poprawnie zakończyć działanie danego procesu.

## Pierwotnie planowana realizacja komunikacji międzyprocesowej

Proces A wysyłający pojedyncze klatki będzie chciał je przekazać do procesów B i C, aby C mógł już załadować obraz u siebie w trakcie przetwarzania go przez B. W związku z tym klatki umieszczane będą w pamięci współdzielonej. Dostęp do niej będzie synchronizowany przy pomocy blokady odczytu/zapisu (ang. read/write lock), dzięki czemu procesy B i C będą mogły równolegle odczytywać obraz. Żeby uniknąć aktywnego oczekiwania na dostępność danych wykorzystane zostaną kolejki komunikatów. W sytuacji kiedy nie mają na czym operować, procesy B i C będą usypiane. Kiedy proces A przetworzy dane, to umieści je w pamięci współdzielonej i wyśle komunikaty do B i C, budząc je.

Komunikacja z B do C będzie przebiegać podobnie, również zostanie wykorzystana pamięć współdzielona i kolejka komunikatów, ale w tej sytuacji jest jeden producent i jeden konsument danych. W związku z tym do synchronizacji dostępu użyjemy prostych mutexów.

Istotne jest również przekazywanie informacji o aktualnie używanym trybie cenzury z procesu D do C. Będzie to realizowane za pomocą osobnej kolejki komunikatów (bez pamięci współdzielonej), ze względu na to, że jest to wiadomość o małym rozmiarze. Usypianie procesu C w oczekiwaniu na wiadomość od D jest oczywiście niepożądane, ponieważ powinien on cały czas nakładać cenzurę na przychodzący obraz, więc będzie on miał oddzielny wątek, który będzie usypiany, a w razie otrzymania komunikatu będzie on zmieniał aktualny tryb cenzury.

W wersji podstawowej w celu użycia wyżej wymienionych mechanizmów komunikacji międzyprocesowej zostaną wykorzystane ich POSIXowe implementacje, głównie ze względu na najwyższą oczekiwaną wydajność. Możliwe jest przetestowanie wydajności dla różnych form komunikacji międzyprocesowej (np. z użyciem biblioteki boost) i ich implementacji.

## Możliwe sposoby testowania systemu i jego konfiguracji

1. Stosujemy kilka rodzajów filtrów. Mierzymy wpływ na wydajność i “na oko” na efektywność cenzury.
2. Sprawdzenie jak rezerwacja rdzeni dla poszczególnych procesów wpływa na wydajność.
3. Testowanie różnych konfiguracji wartości priorytetów procesów. W podstawowej wersji wszystkie priorytety będą jednakowe. Następnie zostanie zmierzona szybkość działania systemu dla różnych heurystycznie wybranych wartości priorytetów. Można również zmierzyć czas wykonywania obliczeń przez poszczególne procesy w danym fragmencie czasu i nadanie priorytetów proporcjonalnie do czasu działania.
4. Dostosowanie liczby klatek na sekundę tak, aby system działał dostatecznie szybko, ale też obraz wynikowy był możliwie płynny.

## Podział pracy

1. Proces D - Tymoteusz Perka
2. Proces A - Paweł Budniak
3. Proces B - Patryk Dobrowolski
4. Proces C - Patryk Dobrowolski, Paweł Budniak
5. Komunikacja między procesami - Tymoteusz Perka
6. Przeprowadzenie testów, dokumentacja końcowa - Paweł Budniak, Patryk Dobrowolski, Tymoteusz Perka

Testy zostaną przeprowadzone przez wszystkich członków zespołu ze względu na różnice w platformach sprzętowych (rozdzielczości/klatkażu kamery, liczbie rdzeni procesora, mocy obliczeniowej), aby upewnić się, że poprawne działanie naszego systemu jest możliwe nie tylko na pojedynczej platformie testowej.

## Praktyczna implementacja

Nasza praktyczna implementacja nie różni się znacząco od tej planowanej. Wszystko poszło zgodnie z planem, poza użyciem blokady zapis/odczyt, która okazuje się być nieobecna w bibliotece boost. Zrezygnowaliśmy także z kolejki komunikatów pomiędzy A i B oraz A i C, gdyż niepotrzebnie zakłócałoby to płynność wyświetlanego obrazu. W związku z tym proces A wstawia klatkę do pamięci współdzielonej, a B i C pobierają go kiedy tylko mogą. Do synchronizacji dostępu do niej użyliśmy zaś zwykłego mutexu. W celu późniejszego odtworzenia klatki z ciągu bajtów zapisanego w pamięci, przekazujemy za pomocą oddzielnej pamięci współdzielonej rozmiar klatki do procesów B i C.

Zgodnie z założeniami, użytkownik może za pomocą interfejsu zapewnianego przez proces D ustawić maksymalną liczbę przetwarzanych klatek na sekundę. W związku z tym, że metoda z OpenCV do ustawiania takiego limitu (VideoCapture::set) jest nieprzenośna i działa różnie w zależności od wykorzystywanej kamery limit ten jest ustalany “manualnie”. To znaczy jest mierzony czas od przetworzenia ostatniej klatki i przy odbiorze kolejnej sprawdzane jest, czy czas ten jest większy od ustalonego czasu na klatkę wynikającego z limitu fps. Jeśli nie jest, to klatka jest pomijana. Rozwiązanie to nie pozwala na dokładne ustawienie przetwarzania dla dowolnego klatkażu, ale pozwala nałożyć nieprzekraczalny limit na liczbę przesyłanych klatek, dzięki czemu system powinien dobrze radzić sobie nawet z kamerą, która dostarcza obraz z bardzo dużą częstotliwością.

W czasie pierwszych testów okazało się, że proces C każdą klatkę przetwarza w ~6 milisekund, podczas gdy procesowi B zajmuje to kilkadziesiąt milisekund, co powodowało częste nie nadążanie cenzury za twarzą. Aby poprawić wrażenia z działania programu, zastosowaliśmy dodatkową synchronizację pomiędzy procesami B i C w postaci kolejki komunikatów. Po starcie systemu, B i C odbierają klatkę od A w podobnym czasie, po czym C usypia się w oczekiwaniu na wiadomość od B, który wysyła ją, kiedy skończy wyszukiwanie twarzy w klatce i wstawi ich współrzędne do pamięci współdzielonej. Dzięki temu C rysuje cenzurę dokładnie na tej samej klatce (lub znajdującej się bardzo blisko), na której B wykrywał twarze. Ta opcja jest konfigurowalna z poziomu pliku include/names.hpp, za pomocą stałej SYNC_BC.

Dzięki tej synchronizacji łatwiejsze stało się także testowanie wydajności naszego systemu. Proces A razem z klatką wstawia także do pamięci czas, w którym została ona pobrana. Proces C natomiast po zakończeniu przetwarzania i wyświetleniu klatki ma możliwość odczytania aktualnego czasu i obliczenia różnicy pomiędzy odbiorem klatki a aktualnym czasem, co dzięki sekwencyjnemu przetwarzaniu daje nam całkowity czas przetwarzania pojedynczej klatki.

Dodaliśmy do procesów A, B oraz C liczenie zajętości czasu procesora w czasie wykonywania. Odbywa się to za pomocą funkcji systemowej times() oraz funkcji obsługi sygnału SIGINT, przesyłanego do procesów A, B i C przez proces D, kiedy użytkownik zażyczy sobie zakończyć program. Procesy po otrzymaniu takiego sygnału obliczą, a następnie wyświetlą na standardowe wyjście swój PID oraz procent zajętości czasu procesora, a następnie normalnie się zakończą.

Program nasz ma także opcję zapisu czasu przetwarzania każdej klatki do pliku, który znajduje się w folderze .../perf_test. Znajdziemy tam także skrypt Pythonowy, który analizuje tenże plik, dając nam pogląd na wydajność systemu w aktualnej konfiguracji. Tryb testowy jest konfigurowalny za pomocą stałej SAVE_PROCESSING_TIME w pliku .../include/names.hpp.

## Kompilacja i uruchamianie

Aby skompilować program, należy wejść z poziomu konsoli do folderu zawierającego projekt, a następnie użyć polecenia:

*cmake .*

Po wykonaniu konfiguracji dokonujemy kompilacji poleceniem:

*make*

A na końcu uruchomieniem procesu D z uprawnieniami administratora, gdyż jest to niezbędne to zmieniania trybów szeregowania na te specjalnego przeznaczenia:

*sudo ./D.out*

## Testy

Wszystkie testy były przeprowadzone na próbie 30-sekundowej, w sensie proces D powoływał do życia procesy, a następnie wywoływał sleep(30), po czym wysyłał sygnał SIGINT do wszystkich swoich dzieci. Testowane ustawienia szeregowania zostały opisane poniżej:

Test 1 - standardowe szeregowanie, wszystkie rdzenie dostępne
Test 2 - szeregowanie standardowe, wszystkie procesy na jednym rdzeniu
Test 3 - szeregowanie FIFO z priorytetem 99, wszystkie rdzenie dostępne
Test 4 - szeregowanie FIFO z priorytetem 99, każdy proces ma swój rdzeń
Test 5 - szeregowanie RR z priorytetem 99, wszystkie rdzenie dostępne

Pierwsza partia testów, bez obciążenia:

**Komputer 1**

| **Ustawienie** | **Liczba klatek** | **Średnia** | **Odchylenie std** | **Mediana** | **Max** | **Min** |
|---|---|---|---|---|---|---|
| Test 1 | 591 | 106.56 | 37.58 | 105 | 161 | 35 |
| Test 2 | 394 | 133.36 | 35.41 | 134 | 192 | 72 |
| Test 3 | 722 | 92.49 | 44.97 | 87 | 361 | 2 |
| Test 4 | 417 | 124.79 | 35.34 | 124 | 191 | 62 |
| Test 5 | 695 | 88.07 | 42.70 | 87 | 294 | 2 |

**Komputer 2**

| **Ustawienie** | **Liczba klatek** | **Średnia** | **Odchylenie std** | **Mediana** | **Max** | **Min** |
|---|---|---|---|---|---|---|
| Test 1 | 1338 | 39.71 | 15.29 | 38 | 97 | 2 |
| Test 2 | 621 | 76.11 | 19.21 | 75 | 112 | 48 |
| Test 3 | 1507 | 62.08 | 161.60 | 24 | 1405 | 0 |
| Test 4 | 518 | 72.94 | 19.14 | 70 | 111 | 43 |
| Test 5 | 1587 | 240.23 | 686.35 | 24 | 3900 | 1 |

**Komputer 3**

| Ustawienie | Liczba klatek | Średnia | Odchylenie std | Mediana | Max | Min |
|---|---|---|---|---|---|---|
| Test 1 | 961 | 45.76 | 10.99 | 46 | 98 | 24 |
| Test 2 | 487 | 75.01 | 10.87 | 75 | 109 | 55 |
| Test 3 | 1042 | 38.66 | 14.68 | 33 | 99 | 1 |
| Test 4 | 476 | 76.97 | 14.22 | 75 | 146 | 55 |
| Test 5 | 1096 | 39.65 | 14.27 | 34 | 94 | 23 |

Oraz druga, z dodatkowym obciążeniem, do którego wykorzystaliśmy program *stress* z ustawieniami: -c 30, co oznacza utworzenie 30 procesów liczących pierwiastki losowych, dużych liczb całkowitych:

**Komputer 1**

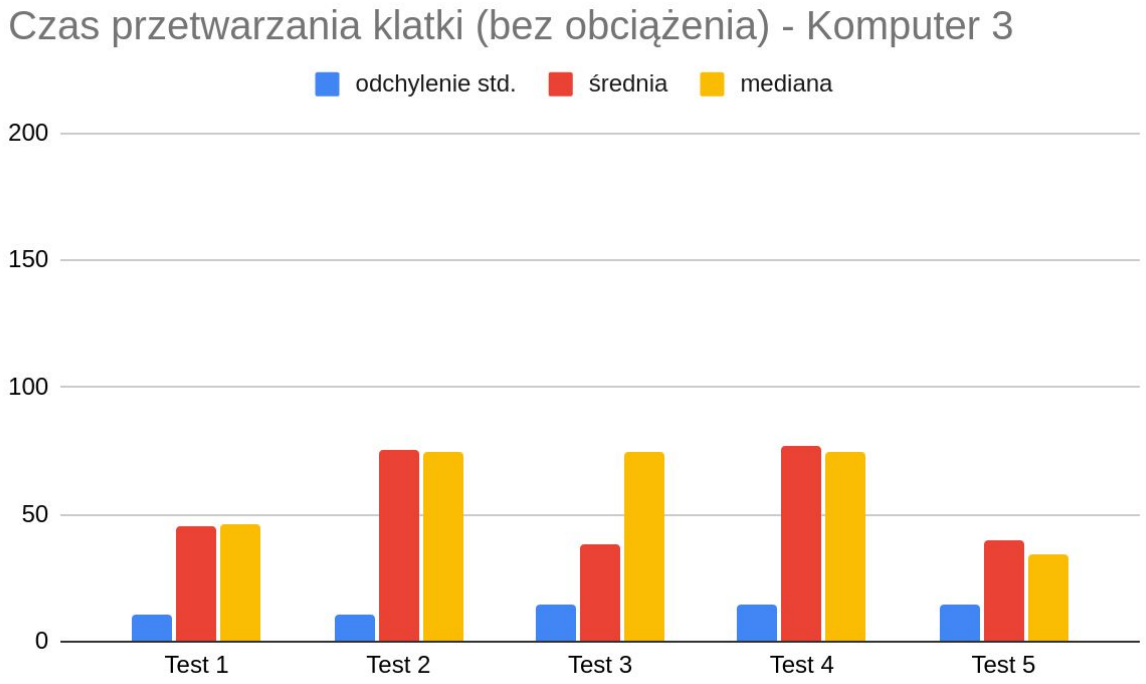
| Ustawienie | Liczba klatek | Średnia | Odchylenie std | Mediana | Max | Min |
|---|---|---|---|---|---|---|
| Test 1 | 250 | 162.21 | 39.78 | 161 | 268 | 79 |
| Test 2 | 187 | 203.56 | 37.37 | 202 | 323 | 127 |
| Test 3 | 691 | 102.60 | 49.46 | 92 | 326 | 2 |
| Test 4 | 99 | 147.07 | 37.39 | 147 | 224 | 75 |
| Test 5 | 694 | 92.73 | 43.77 | 88 | 249 | 3 |

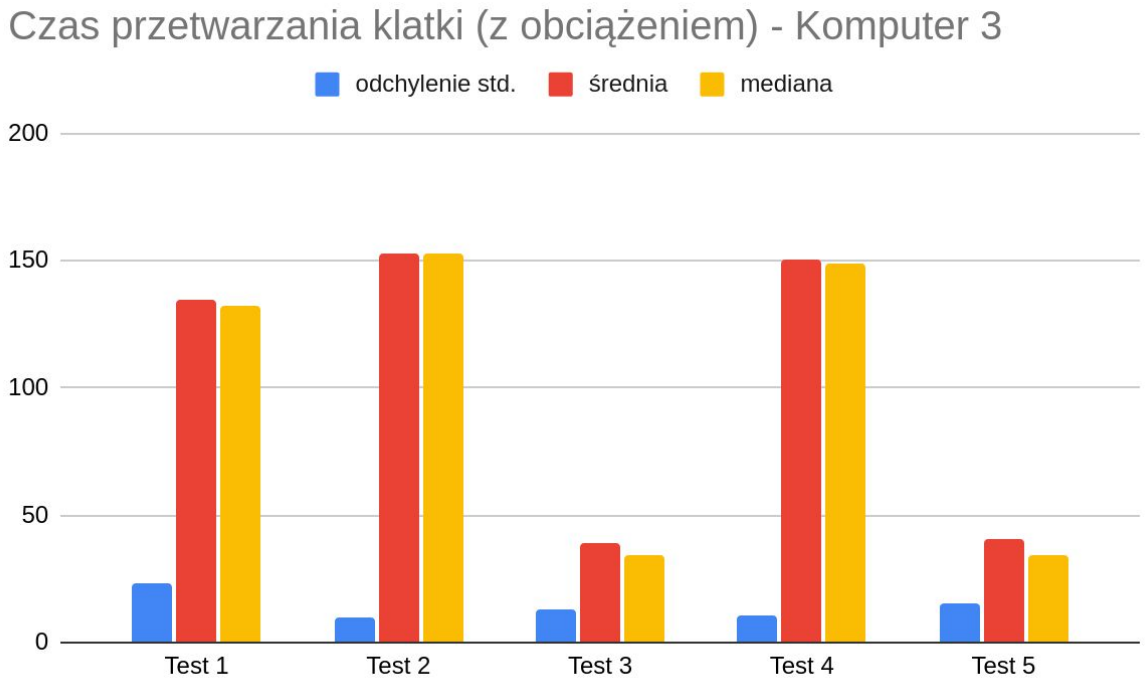
**Komputer 2**

| Ustawienie | Liczba klatek | Średnia | Odchylenie std | Mediana | Max | Min |
|---|---|---|---|---|---|---|
| Test 1 | 608 | 63.89 | 18.03 | 62 | 121 | 27 |
| Test 2 | 635 | 77.24 | 19.34 | 77 | 112 | 42 |
| Test 3 | 1149 | 361.74 | 884.40 | 40 | 4232 | 0 |
| Test 4 | 92 | 76.48 | 20.53 | 80 | 107 | 34 |
| Test 5 | 1639 | 85.36 | 215.00 | 36 | 1574 | 0 |

**Komputer 3**

| Ustawienie | Liczba klatek | Średnia | Odchylenie std | Mediana | Max | Min |
|---|---|---|---|---|---|---|
| Test 1 | 242 | 134.80 | 23.66 | 132 | 210 | 80 |
| Test 2 | 209 | 152.73 | 10.19 | 153 | 181 | 131 |
| Test 3 | 897 | 39.27 | 13.07 | 34 | 108 | 1 |
| Test 4 | 213 | 150.82 | 10.63 | 149 | 181 | 129 |
| Test 5 | 898 | 40.4 | 15.24 | 34 | 168 | 2 |




## Wnioski

Pierwszą naszą obserwacją z testów jest fakt, że proces B potrafi używać wszystkich dostępnych zasobów maszyny i to pomimo faktu, że w naszej implementacji jest to proces jednowątkowy. Okazuje się, że funkcja biblioteki OpenCV, której używamy w naszym systemie do wykrywania twarzy na poszczególnych klatkach, jest dostosowana do możliwości nowoczesnych, wielowątkowych maszyn, gdyż narzędzie *top* użyte na komputerze z 8-wątkowym procesorem nieraz pokazywało

700% utylizacji CPU przez proces B. W związku z tym, że nie mamy wpływu na działanie tej funkcji, ciężko jest znaleźć jakieś sensowne ustawienie szeregowania czy też przydziału procesów do rdzeni, gdyż funkcja ta ma dominujący wpływ na czas przetwarzania klatki (jej wykonanie to około 90% czasu przetwarzania), a nie jesteśmy w stanie przerwać jej wykonania poprzez wywołanie yield(). Także fakt, że procesy pracują w pętli nieskończonej, a nie zaczynają się i kończą utrudnia sensownie zastosowanie trybów szeregowania przeznaczonych do systemów czasu rzeczywistego. Jedynym pomysłem, który okazał się być skuteczny jest ustawienie wszystkich procesów na tryb szeregowania FIFO albo RR z maksymalnym priorytetem, gdyż wtedy mają one pierwszeństwo przed wszystkimi procesami użytkownika, co widać w tabelach, gdzie pomimo obciążenia nie zaobserwujemy dużych zmian w wydajności systemu przy wykorzystaniu tych trybów szeregowania.

W wyniku innych testów ustaliliśmy, że limit fps równy 30 pozwala na płynne wyświetlanie obrazu i dostatecznie szybkie nakładanie cenzury. Zmiana trybu cenzury nie wpływa w żaden znaczący sposób na wydajność. Tryb z filtrem gaussa pozwala na pewną elastyczność w intensywności cenzury za pomocą regulacji parametru sigma. Ponadto wygląda bardziej naturalnie, mniej zakłóca otoczenie. Tryb z zamalowaniem twarzy jednolitym prostokątem oferuje za to skuteczniejszą cenzurę.