Projekt Systemy Operacyjne – Temat 8: SOR

Autor: Hugo Utrata **Numer albumu:** 155238 **Temat:** 8 – SOR

Krótki opis projektu

Projekt realizuje symulację działania Szpitalnego Oddziału Ratunkowego (SOR) z uwzględnieniem rzeczywistych zasad organizacji pracy. System odtwarza przebieg wizyty pacjenta: wejście na SOR (z limitem miejsc), rejestrację, ocenę stanu zdrowia w triażu oraz konsultację u właściwego lekarza specjalisty. Pacjenci mają różne priorytety (czerwony, żółty, zielony) wpływające na kolejność obsługi, a niektórzy mogą być wysyłani do domu już na etapie triażu. Lekarze mogą kierować pacjentów do dalszego leczenia, wypisywać ich do domu lub odsyłać do innej placówki. Symulacja obejmuje również obsługę pacjentów VIP oraz reakcję całego systemu na sygnały Dyrektora (np. przerwanie pracy lekarza, natychmiastowa ewakuacja SOR). Program opiera się na procesach, użyto w nim rozmaite systemy IPC, o których później.

Wytyczne projektu – Temat 8: SOR (Kopia 1:1 PDF)

SOR działa przez całą dobę, zapewniając natychmiastową pomoc osobom w stanach nagłego zagrożenia zdrowia i życia. Działanie SOR-u opiera się na triażu, czyli ocenie stanu pacjentów, który określa priorytet udzielania pomocy (nie decyduje kolejność zgłoszenia). W poczekalni jest N miejsc.

Zasady działania SOR:

- SOR jest czynny całą dobę;
- W poczekalni SOR w danej chwili może się znajdować co najwyżej N pacjentów (pozostali, jeżeli są czekają przed wejściem);
- Dzieci w wieku poniżej 18 lat na SOR przychodzą pod opieką osoby dorosłej;
- Osoby uprawnione VIP (np. honorowy dawca krwi,) do rejestracji podchodzą bez kolejki;
- Każdy pacjent przed wizytą u lekarza musi się udać do rejestracji;
- W przychodni są 2 okienka rejestracji, zawsze działa min. 1 stanowisko;
- Jeżeli w kolejce do rejestracji stoi więcej niż K pacjentów (K>=N/2) otwiera się drugie okienko rejestracji. Drugie okienko zamyka się jeżeli liczba pacjentów w kolejce do rejestracji jest mniejsza niż N/3;

Przebieg wizyty na SOR:

1. Rejestracja:

− Pacjent podaje swoje dane i opisuje objawy.

2. Ocena stanu zdrowia (Triaż):

− Lekarz POZ weryfikuje stan pacjenta i przypisuje mu kolor zgodny z pilnością udzielenia pomocy (na tej podstawie określa się, kto otrzyma pomoc w pierwszej kolejności):

- **czerwony** – natychmiastowa pomoc – ok. 10% pacjentów;
- **żółty** – przypadek pilny – ok. 35% pacjentów;
- **zielony** – przypadek stabilny – ok. 50% pacjentów; − Ok. 5% pacjentów jest odsyłanych do domu bezpośrednio z triażu; − Lekarz POZ po przypisaniu koloru, kieruje danego pacjenta do lekarza specjalisty: kardiologa, neurologa, okulisty, laryngologa, chirurga, pediatry.

3. Wstępna diagnostyka i leczenie:

− Lekarz specjalista wykonuje niezbędne badania (wywiad, badanie fizykalne, EKG, pomiar ciśnienia, …), aby ustabilizować funkcje życiowe pacjenta.

4. Decyzja o dalszym postępowaniu:

− Po wstępnej diagnozie i stabilizacji stanu pacjent może zostać przez lekarza specjalistę:

- wypisany do domu – ok. 85% pacjentów;
- skierowany na dalsze leczenie do odpowiedniego oddziału szpitalnego – ok. 14.5% pacjentów;
- skierowany do innej, specjalistycznej placówki – ok. 0,5% pacjentów.

Sygnały Dyrektora:

- **Sygnał 1** – dany lekarz specjalista bada bieżącego pacjenta i przerywa pracę na SOR-rze i udaje się na oddział. Wraca po określonym losowo czasie.
- **Sygnał 2** – wszyscy pacjenci i lekarze natychmiast opuszczają budynek.

ARCHITEKTURA IPC

A. Tworzenie i zarządzanie procesami (Generator Pacjentów)

Generator pacjentów (`generuj.c`) jest odpowiedzialny za dynamiczne tworzenie procesów pacjentów w kontrolowany sposób, zapobiegając jednocześnie

przeciążeniu system przez fork - bomby.

A.1. Ograniczanie liczby jednoczesnych procesów

Przed każdym wywołaniem `fork()`, generator wykonuje operację P na semaforze `SEM_GENERATOR`, który jest inicjalizowany wartością `MAX_PROCESOW` (wartość wedle uznania, ogranicza nas systemowy max dla semafora). Jeśli limit jest wyczerpany, proces generatora blokuje się do momentu zwolnienia miejsca. forkowanie i semop: generuj.c

```
// generuj.c - pętla główna
struct sembuf zajmij = {SEM_GENERATOR, -1, SEM_UNDO};
if (semop(semid, &zajmij, 1) == -1) {
    if (errno == EINTR) { if (ewakuacja) break; i--; continue; }
    break;
}

pid_t pid = fork();
if (pid == 0) {
    execl("./pacjent", "pacjent", NULL);
    exit(1);
}
```

Flaga `SEM_UNDO` zapewnia automatyczne zwolnienie semafora w przypadku nieoczekiwanego zakończenia procesu generatora.

A.2. Obsługa zakończenia procesów potomnych (SIGCHLD)

Kiedy proces pacjenta kończy działanie (przez `exit()` lub sygnał), jądro wysyła sygnał `SIGCHLD` do procesu rodzica (generatora). Handler ustawia flagę `sigchld_received`, a właściwa obsługa odbywa się w funkcji `zbierz_zombie()`:

zbieranie zombie: generuj.c

```
// generuj.c - handler i obsługa SIGCHLD
volatile sig_atomic_t sigchld_received = 0;

void handle_sigchld(int sig) { sigchld_received = 1; }

void zbierz_zombie() {
    int status;
    pid_t pid;
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        if (semid != -1) {
            struct sembuf unlock = {SEM_GENERATOR, 1, SEM_UNDO};
            semop(semid, &unlock, 1);
        }
    }
    sigchld_received = 0;
}
```

Funkcja `waitpid(-1, &status, WNOHANG)` zbiera statusy zakończonych dzieci bez blokowania. Dla każdego zebranego procesu, semafor `SEM_GENERATOR` jest podnoszony (+1), zwalniając miejsce dla nowego pacjenta.

A.3. Konfiguracja sigaction

Struktura `sigaction` jest konfigurowana z kluczowymi flagami:

struktura sigaction dla SIGCHLD: generuj.c

```
// generuj.c - konfiguracja sygnałów
struct sigaction sa;
sa.sa_handler = handle_sigchld;
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;
sigaction(SIGCHLD, &sa, NULL);
```

- **`SA_RESTART`**: Po obsłużeniu sygnału `SIGCHLD`, przerwane wywołanie systemowe (np. `semop`) zostanie automatycznie wznowione zamiast zwracać błąd `EINTR`.
- **`SA_NOCLDSTOP`**: Sygnał `SIGCHLD` będzie wysyłany tylko przy faktycznym zakończeniu procesu potomnego, nie przy jego zatrzymaniu (`SIGSTOP`).

A.4. Podmiana obrazu procesu (execl)

Po udanym `fork()`, proces potomny natychmiast podmienia swój obraz na program pacjenta za pomocą `execl()`. Przekazanie argumentów przez `argv` nie jest tutaj wykorzystywane - używany go w innych częściach systemu (np. przy uruchamianiu lekarzy).

B. Pamięć dzielona i synchronizacja (Struktura StanSOR)

Centralna struktura `StanSOR` w pamięci dzielonej przechowuje globalny stan systemu, dostępny dla wszystkich procesów.

B.1. Definicja struktury StanSOR

definicja StanSor: wspolne.h

```
// wspolne.h - struktura pamięci dzielonej
typedef struct
{
    int symulacja_trwa;              // Flaga aktywności symulacji
    int dostepni_specjalisci[7];     // Status dostępności lekarzy [0-6]
    int dlugosc_kolejki_rejestracji; // Aktualna długość kolejki
    int czy_okienko_2_otwarte;       // Status drugiego okienka (0/1)

    int pacjenci_przed_sor;          // Liczba osób czekających na wejście
    int pacjenci_w_poczekalni;       // Liczba osób wewnątrz SOR

    int wymuszenie_otwarcia;         // Rozkaz otwarcia/zamknięcia okienka 2

    int snap_w_srodku;               // Snapshot dla ewakuacji
    int snap_przed_sor;              // Snapshot dla ewakuacji
} StanSOR;
```

B.2. Inicjalizacja pamięci dzielonej

Pamięć dzielona jest tworzona i inicjalizowana w procesie `main`:

inicjalizacja shm w main: main.c

```
// main.c - tworzenie i inicjalizacja
key_t key_shm = ftok(FILE_KEY, ID_SHM_MEM);
shmid = shmget(key_shm, sizeof(StanSOR), IPC_CREAT | 0600);
StanSOR *stan = (StanSOR*)shmat(shmid, NULL, 0);
memset(stan, 0, sizeof(StanSOR));
stan->symulacja_trwa = 1;
for (int i = 1; i <= 6; i++) stan->dostepni_specjalisci[i] = 1;
shmdt(stan);
```

Wszystkie pola są zerowane przez `memset()`, po czym ustawiane są wartości początkowe: flaga symulacji oraz dostępność wszystkich specjalistów.

B.3. Ochrona dostępu mutexem (SEM_DOSTEP_PAMIEC)

Każda* operacja na strukturze `StanSOR` musi być otoczona sekcją krytyczną z wykorzystaniem semafora `SEM_DOSTEP_PAMIEC`: przyklad dla pacjenta: pacjent.c

```
// pacjent.c - funkcja aktualizuj_liczniki()
void aktualizuj_liczniki(int zmiana_przed, int zmiana_wew, int zmiana_kolejki_rej) {
    StanSOR *stan = (StanSOR*)shmat(shmid, NULL, 0);
    if (stan != (void*)-1) {
        struct sembuf lock = {SEM_DOSTEP_PAMIEC, -1, SEM_UNDO};
        struct sembuf unlock = {SEM_DOSTEP_PAMIEC, 1, SEM_UNDO};

        while(semop(semid, &lock, 1) == -1) {
            if (errno == EINTR) continue;
            return;
        }

        // Modyfikacje struktury StanSOR
        stan->pacjenci_przed_sor += zmiana_przed;
        stan->pacjenci_w_poczekalni += zmiana_wew;
        stan->dlugosc_kolejki_rejestracji += zmiana_kolejki_rej;

        // ... logika progów otwarcia/zamknięcia okienka ...

        semop(semid, &unlock, 1);
        shmdt(stan);
    }
}
```

Wzorzec `while(semop(...) == -1) { if (errno == EINTR) continue; }` zapewnia poprawną obsługę przerwań sygnałami podczas oczekiwania na semafor.

C. Dynamiczne zarządzanie okienkami rejestracji

System implementuje dynamiczne otwieranie i zamykanie drugiego okienka rejestracji w oparciu o aktualne obciążenie kolejki.

C.1. Progi otwarcia i zamknięcia

Progi są zdefiniowane jako makra w `wspolne.h`:

```
// wspolne.h - definicje progów
#define PROG_OTWARCIA (MAX_PACJENTOW / 2)   // >= 400 dla MAX_PACJENTOW=800
#define PROG_ZAMKNIECIA (MAX_PACJENTOW / 3) // < 266 dla MAX_PACJENTOW=800
```

Rożne progi zwalniające zapobiegają oscylacji stanu przy granicznym obciążeniu.

C.2. Detekcja progu przez proces pacjenta

Proces pacjenta przy wejściu do poczekalni sprawdza długość kolejki i ustawia flagę `wymuszenie_otwarcia` w pamięci dzielonej:

flaga w shm: pacjent.c

```
// pacjent.c - fragment aktualizuj_liczniki()
int q = stan->dlugosc_kolejki_rejestracji;
int cmd = stan->wymuszenie_otwarcia;

if (q > PROG_OTWARCIA && cmd == 0)
{
    stan->wymuszenie_otwarcia = 1;
    zapisz_raport(RAPORT_1, semid, "[ Pacjent ] wymuszam otwarcie bramki numer 2\n");
}
else if (q < PROG_ZAMKNIECIA && cmd == 1)
{
    stan->wymuszenie_otwarcia = 0;
    zapisz_raport(RAPORT_1, semid, "[ Pacjent ] wymuszam zamkniecie bramki numer 2\n");
}
```

Przy 10 000 procesów istnieje 99.99% szans, że to pacjent pierwszy wykryje potrzebę otwarcia drugiej bramki - korzystamy z tego, że akurat wchodzi do pamięci dzielonej chronionej mutexem. Pełni on rolę czujnika - gdy nadejdzie potrzeba ustawia flage wymuszenia otwarcia.

C.3. Wątek bramki w procesie main (watek_bramka)

Właściwe sterowanie okienkiem odbywa się w dedykowanym wątku `watek_bramka` działającym w procesie `main`. Wątek sprawdza flagę `wymuszenie_otwarcia` i wykonuje odpowiednie akcje:

watek_bramka: main.c

```
// main.c - wątek watek_bramka()
void* watek_bramka(void* arg)
{
    StanSOR *stan = (StanSOR*)shmat(shmid, NULL, 0);
    if (stan == (void*)-1) return NULL;

    int local_okienko_otwarte = 0;

    while (monitor_running) {
        usleep(5000); // 5ms - szybka reakcja

        int rozkaz = stan->wymuszenie_otwarcia;

        // Synchronizacja stanu dla innych procesów
        if (stan->czy_okienko_2_otwarte != local_okienko_otwarte) {
             stan->czy_okienko_2_otwarte = local_okienko_otwarte;
        }

        if (!local_okienko_otwarte && rozkaz == 1) {
            // OTWARCIE: fork + execl nowego procesu rejestracji
            pid_t pid = fork();
            if (pid == 0) {
                signal(SIGTERM, SIG_DFL);
                execl("./rejestracja", "SOR_rejestracja", "2", NULL);
                exit(1);
            } else if (pid > 0) {
                pid_rejestracja_2 = pid;
                local_okienko_otwarte = 1;
                stan->czy_okienko_2_otwarte = 1;
                zapisz_raport(RAPORT_1, semid, "[MONITOR] Otwieram bramke nr 2\n");
            }
        }
        else if (local_okienko_otwarte && rozkaz == 0) {
            // ZAMKNIĘCIE: wysłanie SIGINT i oczekiwanie na zakończenie
            if (pid_rejestracja_2 > 0) {
                kill(pid_rejestracja_2, SIGINT);
                waitpid(pid_rejestracja_2, NULL, 0);
                pid_rejestracja_2 = -1;
                local_okienko_otwarte = 0;
                stan->czy_okienko_2_otwarte = 0;
                zapisz_raport(RAPORT_1, semid, "[MONITOR] Zamykam bramke nr 2\n");
            }
        }
    }
    // Sprzątanie przy zakończeniu wątku
    if (local_okienko_otwarte && pid_rejestracja_2 > 0) {
        kill(pid_rejestracja_2, SIGTERM);
        waitpid(pid_rejestracja_2, NULL, 0);
    }
    shmdt(stan);
    return NULL;
}
```

Co ważne, wątek nie wchodzi do pamięci dzielonej nie korzystając z mutexu - flaga otwarcia drugiej bramki jest atomowa 0/1 - przez to reakcja jest niemalże natychmiastowa.

Wątek utrzymuje lokalny stan `local_okienko_otwarte`, który jest synchronizowany z pamięcią dzieloną. Dzięki temu inne procesy mogą odczytać aktualny status okienka.

C.4. Proces rejestracji

Proces rejestracji (`rejestracja.c`) w pętli odbiera komunikaty od pacjentów i odsyła je z powrotem (z ustawionym `mtype` na PID pacjenta). Zawiera trzy istotne mechanizmy bezpieczeństwa:

1. **Weryfikacja istnienia pacjenta** – przed wysłaniem odpowiedzi, rejestracja sprawdza za pomocą `kill(pid, 0)` czy proces pacjenta nadal żyje. Jeśli pacjent zakończył działanie (np. przez `SEM_UNDO` po sygnale), wiadomość jest anulowana zamiast zapychać kolejkę martwymi odpowiedziami.
2. **Dynamiczny bufor przepełnienia** – wysyłanie odpowiedzi odbywa się z flagą `IPC_NOWAIT` . Gdy kolejka jest pełna (`EAGAIN`), zamiast blokowania, pacjent trafia do dynamicznego bufora w pamięci procesu. Bufor jest alokowany na starcie na podstawie rzeczywistej pojemności kolejki systemowej (`msg_qbytes / sizeof_msg`).
3. **Opróżnianie bufora** – na początku każdej iteracji pętli, jeśli bufor zawiera oczekujące wiadomości, rejestracja sprawdza stan kolejki przez `msgctl(IPC_STAT)` i próbuje odesłać zbuforowane komunikaty dopóki kolejka ma wolne miejsce.

główna pętla rejestracji: rejestracja.c

```
// rejestracja.c - inicjalizacja bufora
struct msqid_ds stan_kolejki;
pobierz_stan_kolejki(msgid_we, &stan_kolejki);
size_t rozmiar_pojedynczej_wiadomosci = sizeof(KomunikatPacjenta) - sizeof(long);
int max_msg_limit = stan_kolejki.msg_qbytes / rozmiar_pojedynczej_wiadomosci;

bufor_pojemnosc = max_msg_limit;
bufor_oczekujacych = (KomunikatPacjenta*)malloc(bufor_pojemnosc * sizeof(KomunikatPacjenta));
```

```
// rejestracja.c - główna pętla z buforem i weryfikacją pacjenta
while(!koniec_pracy)
{
    pobierz_stan_kolejki(msgid_we, &stan_kolejki);

    // Opróżnianie bufora gdy kolejka ma wolne miejsce
    if (bufor_licznik > 0) {
        if (stan_kolejki.msg_qnum < max_msg_limit) {
            for (int i = 0; i < bufor_licznik; i++) {
                if (msgsnd(msgid_we, &bufor_oczekujacych[i], rozmiar_pojedynczej_wiadomosci, IPC_NOWAIT) != -1) {
                    zapisz_raport(KONSOLA, semid, "[ Rejestracja %d ] Wznowiono z bufora: %d\n",
                                nr_okienka, bufor_oczekujacych[i].pacjent_pid);
                    for(int j=i; j<bufor_licznik-1; j++) bufor_oczekujacych[j] = bufor_oczekujacych[j+1];
                    bufor_licznik--;
                    i--;
                } else {
                    if (errno == EAGAIN) break;
                }
            }
        }
    }

    ssize_t status = msgrcv(msgid_we, &pacjent, rozmiar_pojedynczej_wiadomosci, -2, IPC_NOWAIT);
    if (status == -1) {
        if (errno == ENOMSG || errno == EINTR) { usleep(50000); continue; }
        break;
    }

    pacjent.mtype = pacjent.pacjent_pid;

    // Weryfikacja czy pacjent nadal żyje
    if (kill(pacjent.pacjent_pid, 0) == -1 && errno == ESRCH) {
        zapisz_raport(KONSOLA, semid, "[ Rejestracja %d ] Brak informacji o pacjencie %d (exit), Anuluje wysylanie wiadomosci\n",
                     nr_okienka, pacjent.pacjent_pid);
        continue;
    }

    // Wysyłanie z IPC_NOWAIT - przy pełnej kolejce pacjent trafia do bufora
    if(msgsnd(msgid_we, &pacjent, rozmiar_pojedynczej_wiadomosci, IPC_NOWAIT) != -1) {
        if(!koniec_pracy) {
            zapisz_raport(KONSOLA, semid, "[ Rejestracja %d ] Pacjent %d -> POZ\n", nr_okienka, pacjent.pacjent_pid);
        }
    }
    else if (errno == EAGAIN) {
        if (bufor_licznik < bufor_pojemnosc) {
            bufor_oczekujacych[bufor_licznik++] = pacjent;
            zapisz_raport(KONSOLA, semid, "[ Rejestracja %d ] KOLEJKA FULL Pacjent %d -> BUFOR (%d/%d).\n",
                          nr_okienka, pacjent.pacjent_pid, bufor_licznik, bufor_pojemnosc);
        } else {
            zapisz_raport(KONSOLA, semid, "[CRITICAL] BUFOR PRZEPELNIONY! Pacjent %d porzucony.\n", pacjent.pacjent_pid);
        }
    }
}
if (bufor_oczekujacych) free(bufor_oczekujacych);
```

Flaga `IPC_NOWAIT` w `msgrcv()` sprawia, że proces nie blokuje się gdy kolejka jest pusta – zamiast tego zwraca błąd `ENOMSG`, co pozwala na responsywne sprawdzanie flagi `koniec_pracy`.

Ujemna wartość `-2` w argumencie `msgtype` oznacza odbiór wiadomości o typie <= 2 z priorytetem dla mniejszych wartości (VIP=1 przed Zwykły=2).

D. System kolejek komunikatów

D.1. Struktura komunikatu pacjenta

struktura dla kolejek: wspolne.h

```
// wspolne.h - struktura komunikatu
typedef struct {
    long mtype;        // Typ wiadomości (priorytet/PID)
    pid_t pacjent_pid; // PID procesu pacjenta
    int typ_lekarza;   // Docelowy specjalista (0=POZ, 1-6=specjaliści)
    int czy_vip;       // Flaga VIP
    int wiek;          // Wiek pacjenta
    int kolor;         // Kolor triażu (1=czerwony, 2=żółty, 3=zielony)
    int skierowanie;   // Decyzja końcowa (1=dom, 2=oddział, 3=inna placówka)
} KomunikatPacjenta;
```

Pole `mtype` pełni podwójną rolę:

- **Przy wysyłaniu DO lekarza**: oznacza priorytet (1=VIP/Czerwony, 2=Żółty/Zwykły, 3=Zielony)
- **Przy odsyłaniu DO pacjenta**: przyjmuje wartość `pacjent_pid`, umożliwiając precyzyjne adresowanie

D.2. Tworzenie kolejek komunikatów

Wszystkie kolejki są tworzone w procesie `main` za pomocą funkcji pomocniczej:

msg_creat: main.c

```
// main.c - tworzenie kolejek
int msg_creat(int index, int klucz) {
    int id = msgget(ftok(FILE_KEY, klucz), IPC_CREAT | 0600);
    msgs_ids[index] = id;
    return id;
}

// Wywołania w main():
msg_creat(0, ID_KOLEJKA_REJESTRACJA);  // 'R'
msg_creat(1, ID_KOLEJKA_POZ);          // 'P'
msg_creat(2, ID_KOL_KARDIOLOG);        // 'K'
msg_creat(3, ID_KOL_NEUROLOG);         // 'N'
msg_creat(4, ID_KOL_LARYNGOLOG);       // 'L'
msg_creat(5, ID_KOL_CHIRURG);          // 'C'
msg_creat(6, ID_KOL_OKULISTA);         // 'O'
msg_creat(7, ID_KOL_PEDIATRA);         // 'D'
```

D.3. Przepływ komunikatów i cykl priorytetów

Komunikat pacjenta przechodzi przez system w następujący sposób:

```
PACJENT -> REJESTRACJA -> PACJENT -> POZ -> PACJENT -> SPECJALISTA -> PACJENT
```

Na każdym etapie `mtype` jest odpowiednio modyfikowany. Dorosły pacjent wykonuje operacje IPC samodzielnie za pomocą funkcji `wykonaj_ipc_samodzielnie()`, natomiast dziecko deleguje je do wątku opiekuna (szczegóły w późniejszej sekcji):

```
// pacjent.c - uniwersalna funkcja IPC dla dorosłych pacjentów
void wykonaj_ipc_samodzielnie(int qid, int limit_id, KomunikatPacjenta *msg) {
    lock_limit(limit_id);
    while (msgsnd(qid, msg, sizeof(KomunikatPacjenta)-sizeof(long), 0) == -1) if(errno!=EINTR) break;
    while (msgrcv(qid, msg, sizeof(KomunikatPacjenta)-sizeof(long), msg->pacjent_pid, 0) == -1) if(errno!=EINTR) break;
    unlock_limit(limit_id);
}
```

Funkcja ta opakowuje pełen cykl komunikacji: zajęcie semafora limitującego, wysłanie wiadomości, oczekiwanie na odpowiedź adresowaną po PID, zwolnienie semafora. Dzięki temu logika IPC jest zunifikowana i nie powtarza się w kodzie dla każdego etapu wizyty.

```
// pacjent.c - sekwencja komunikacji dorosłego pacjenta
msg.mtype = vip ? TYP_VIP : TYP_ZWYKLY;  // 1 lub 2
wykonaj_ipc_samodzielnie(rej_msgid, SLIMIT_REJESTRACJA, &msg);

msg.mtype = 1;  // POZ nie rozróżnia priorytetów
wykonaj_ipc_samodzielnie(poz_id, SLIMIT_POZ, &msg);

if (msg.typ_lekarza > 0) {
    int spec_id = msg.typ_lekarza;
    int qid = msgget(ftok(FILE_KEY, (spec_id==1?'K':spec_id==2?'N':spec_id==3?'L':
                                     spec_id==4?'C':spec_id==5?'O':'D')), 0);
    msg.mtype = msg.kolor;  // 1=czerwony (najwyższy), 2=żółty, 3=zielony
    wykonaj_ipc_samodzielnie(qid, spec_id + 1, &msg);
}
```

D.4. Odbiór z priorytetem (msgrcv z ujemnym mtype)

Lekarze specjaliści odbierają wiadomości z priorytetem dla pilniejszych przypadków:

przykład dla lekarza specjalisty: lekarz.c

```
// lekarz.c - odbiór przez specjalistę
if(msgrcv(msgid, &pacjent, sizeof(pacjent) - sizeof(long), -3, IPC_NOWAIT) == -1) {
    // ...
}
```

Wartość `-3` oznacza: "odbierz wiadomość o typie <= 3, wybierając najpierw tę z najmniejszym typem". Skutkuje to obsługą w kolejności: Czerwony (1) → Żółty (2) → Zielony (3).

D.5. Limitowanie kolejek (semafory SLIMIT)

Aby zapobiec przepełnieniu systemowych buforów kolejek komunikatów, wprowadzono mechanizm ograniczania producenta:

```
// pacjent.c - funkcje limitujące
void lock_limit(int sem_indeks) {
    struct sembuf operacja = {sem_indeks, -1, SEM_UNDO};
    while (semop(semid_limits, &operacja, 1) == -1) {
        if(errno == EINTR) continue;
        break;
    }
}

void unlock_limit(int sem_indeks) {
    struct sembuf operacja = {sem_indeks, 1, SEM_UNDO};
    semop(semid_limits, &operacja, 1);
}
```

Semafor jest zwalniany dopiero po odebraniu odpowiedzi od lekarza, co gwarantuje, że w każdej kolejce nigdy nie będzie więcej niż `INT_LIMIT_KOLEJEK` oczekujących komunikatów.

E. Logika lekarzy (lekarz.c)

E.1. Rozróżnienie typów lekarzy

Typ lekarza jest przekazywany jako argument przy uruchomieniu procesu:

uruchamianie lekarzy

```
// main.c - uruchamianie lekarzy
pid_poz = uruchom_proces("./lekarz", "SOR_POZ", "0");  // POZ
for(int i=1; i<=6; i++) {
    char buff[5]; sprintf(buff, "%d", i);
    pid_lekarze[i] = uruchom_proces("./lekarz", nazwy_lek[i], buff);
}

// lekarz.c - odczyt typu
typ_lekarza = atoi(argv[1]);

if (typ_lekarza == 0) praca_poz(msgid_poz);
else {
    int symbol = (typ_lekarza==1?'K':typ_lekarza==2?'N':typ_lekarza==3?'L':
                  typ_lekarza==4?'C':typ_lekarza==5?'O':'D');
    int mid = msgget(ftok(FILE_KEY, symbol), 0);
    praca_specjalista(typ_lekarza, mid);
}
```

E.2. Praca lekarza POZ (Triaż)

Lekarz POZ przypisuje pacjentowi kolor triażu i kieruje do odpowiedniego specjalisty. Analogicznie do rejestracji, POZ posiada mechanizm weryfikacji istnienia pacjenta oraz dynamiczny bufor przepełnienia:

pętla główna lekarz POZ: lekarz.c

```
// lekarz.c - inicjalizacja bufora w praca_poz()
struct msqid_ds stan_kolejki;
pobierz_stan_kolejki(msgid_poz, &stan_kolejki);
size_t rozmiar_msg = sizeof(KomunikatPacjenta) - sizeof(long);
int max_msg_limit = stan_kolejki.msg_qbytes / rozmiar_msg;

KomunikatPacjenta *bufor = (KomunikatPacjenta*)malloc(max_msg_limit * sizeof(KomunikatPacjenta));
int bufor_licznik = 0;
int bufor_pojemnosc = max_msg_limit;
```

```
// lekarz.c - funkcja praca_poz() - główna pętla
while(!koniec_pracy)
{
    // Opróżnianie bufora gdy kolejka ma wolne miejsce
    if (bufor_licznik > 0) {
        pobierz_stan_kolejki(msgid_poz, &stan_kolejki);
        if (stan_kolejki.msg_qnum < max_msg_limit) {
            for (int i = 0; i < bufor_licznik; i++) {
                if (msgsnd(msgid_poz, &bufor[i], rozmiar_msg, IPC_NOWAIT) != -1) {
                    zapisz_raport(KONSOLA, semid, "[POZ] Wznowiono pacjenta z bufora: %d\n", bufor[i].pacjent_pid);
                    for(int j=i; j<bufor_licznik-1; j++) bufor[j] = bufor[j+1];
                    bufor_licznik--;
                    i--;
                } else {
                    if (errno == EAGAIN) break;
                }
            }
        }
    }

    if(msgrcv(msgid_poz, &pacjent, rozmiar_msg, -1, IPC_NOWAIT) == -1)
    {
        if (errno == ENOMSG || errno == EINTR) { usleep(50000); continue; }
        break;
    }

    // Losowanie koloru zgodnie z rozkładem statystycznym
    int r = rand() % 1000;
    if (r < 100) pacjent.kolor = CZERWONY;       // 10%
    else if (r < 450) pacjent.kolor = ZOLTY;     // 35%
```

```
    else if (r < 950) pacjent.kolor = ZIELONY;   // 50%
    else {
        // 5% - odesłanie do domu
        pacjent.typ_lekarza = 0;
        pacjent.skierowanie = 1;
        pacjent.kolor = 0;
    }

    // Przypisanie specjalisty
    if (pacjent.kolor) {
        if (pacjent.wiek < 18) pacjent.typ_lekarza = LEK_PEDIATRA;  // Dzieci do pediatry
        else pacjent.typ_lekarza = (rand() % 5) + 1;  // Dorośli losowo 1-5
    }

    pacjent.mtype = pacjent.pacjent_pid;
    if(koniec_pracy) break;

    // Weryfikacja czy pacjent nadal żyje
    if (kill(pacjent.pacjent_pid, 0) == -1 && errno == ESRCH) {
        zapisz_raport(KONSOLA, semid, "[ POZ ] Brak informacji o pacjencie %d (exit), Anuluje wysylanie wiadomosci\n",
                    pacjent.pacjent_pid);
        continue;
    }

    // Wysyłanie z IPC_NOWAIT - przy pełnej kolejce pacjent trafia do bufora
    if (msgsnd(msgid_poz, &pacjent, rozmiar_msg, IPC_NOWAIT) == -1) {
        if (errno == EAGAIN) {
            if (bufor_licznik < bufor_pojemnosc) {
                bufor[bufor_licznik++] = pacjent;
                zapisz_raport(KONSOLA, semid, "[POZ] KOLEJKA PELNA! Pacjent %d -> BUFOR (%d/%d)\n",
                            pacjent.pacjent_pid, bufor_licznik, bufor_pojemnosc);
            } else {
                zapisz_raport(KONSOLA, semid, "[POZ] CRITICAL: Bufor przepelniony. Pacjent %d porzucony.\n",
                            pacjent.pacjent_pid);
            }
        }
    }
}
free(bufor);
```

Własna interpretacja: Przy odesłaniu pacjenta przez POZ do domu nadajemy kolor 0 - niezdefiniowany. Uznaję tym samym, że pacjent kończy tutaj swoje badania i zwyczajnie w jego przypadku priorytet jest nieistotny (zdrowy).

E.3. Praca lekarza specjalisty

Specjalista podejmuje końcową decyzję o dalszym postępowaniu. Identycznie jak POZ i rejestracja, specjalista posiada mechanizm weryfikacji istnienia pacjenta (`kill(pid, 0)`) oraz dynamiczny bufor przepełnienia:

pętla główna lekarza specjalisty: lekarz.c

```
// lekarz.c - funkcja praca_specjalista() z buforem i weryfikacją
void praca_specjalista(int typ, int msgid)
{
    StanSOR *stan = (StanSOR*)shmat(shmid, NULL, 0);
    // ...

    // Inicjalizacja bufora na podstawie pojemności kolejki
    struct msqid_ds stan_kolejki;
    pobierz_stan_kolejki(msgid, &stan_kolejki);
    size_t rozmiar_msg = sizeof(KomunikatPacjenta) - sizeof(long);
    int max_msg_limit = stan_kolejki.msg_qbytes / rozmiar_msg;

    KomunikatPacjenta *bufor = (KomunikatPacjenta*)malloc(max_msg_limit * sizeof(KomunikatPacjenta));
    int bufor_licznik = 0;
    int bufor_pojemnosc = max_msg_limit;

    while(!koniec_pracy)
```

```
    while(!koniec_pracy)
    {
        // Obsługa wezwania na oddział (szczegóły w sekcji F)
        if(wezwanie_na_oddzial) { /* ... */ }

        // Opróżnianie bufora gdy kolejka ma wolne miejsce
        if (bufor_licznik > 0) {
            pobierz_stan_kolejki(msgid, &stan_kolejki);
            if (stan_kolejki.msg_qnum < max_msg_limit) {
                for (int i = 0; i < bufor_licznik; i++) {
                    if (msgsnd(msgid, &bufor[i], rozmiar_msg, IPC_NOWAIT) != -1) {
                        zapisz_raport(KONSOLA, semid, "[%s] Wznowiono z bufora: %d\n",
                                    int_to_lekarz(typ), bufor[i].pacjent_pid);
                        for(int j=i; j<bufor_licznik-1; j++) bufor[j] = bufor[j+1];
                        bufor_licznik--;
                        i--;
                    } else {
                        if (errno == EAGAIN) break;
                    }
                }
            }
        }

        // Odbiór pacjenta z priorytetem
        if(msgrcv(msgid, &pacjent, rozmiar_msg, -3, IPC_NOWAIT) == -1) {
            if (errno == ENOMSG || errno == EINTR) { usleep(50000); continue; }
            break;
        }

        // Decyzja końcowa zgodnie z rozkładem
        int r = rand() % 1000;
        if (r < 850) pacjent.skierowanie = 1;       // 85% - do domu
        else if (r < 995) pacjent.skierowanie = 2;  // 14.5% - na oddział
        else pacjent.skierowanie = 3;               // 0.5% - inna placówka

        pacjent.mtype = pacjent.pacjent_pid;

        // Weryfikacja czy pacjent nadal żyje
        if (kill(pacjent.pacjent_pid, 0) == -1 && errno == ESRCH) {
            zapisz_raport(KONSOLA, semid, "[ %s ] Brak informacji o pacjencie %d. Anuluje wiadomość zwrotną\n",
                         int_to_lekarz(typ), pacjent.pacjent_pid);
            continue;
        }

        // Wysyłanie z IPC_NOWAIT - przy pełnej kolejce pacjent trafia do bufora
        if (msgsnd(msgid, &pacjent, rozmiar_msg, IPC_NOWAIT) == -1) {
             if (errno == EAGAIN) {
                if (bufor_licznik < bufor_pojemnosc) {
                    bufor[bufor_licznik++] = pacjent;
                    zapisz_raport(KONSOLA, semid, "[%s] KOLEJKA PELNA! Pacjent %d -> BUFOR (%d/%d)\n",
                                int_to_lekarz(typ), pacjent.pacjent_pid, bufor_licznik, bufor_pojemnosc);
                } else {
                    zapisz_raport(KONSOLA, semid, "[%s] CRITICAL: Bufor przepelniony. Pacjent %d porzucony.\n",
                                int_to_lekarz(typ), pacjent.pacjent_pid);
                }
            }
        }
    }

    free(bufor);
    shmdt(stan);
}
```

Podsumowanie mechanizmów bezpieczeństwa w procesach SOR (rejestracja, POZ, specjaliści):

Wszystkie trzy typy procesów obsługujących pacjentów implementują identyczny wzorzec ochronny:

1. **`kill(pacjent_pid, 0)` przed `msgsnd()`** – sprawdzenie czy proces pacjenta istnieje. Eliminuje wysyłanie odpowiedzi do martwych procesów, co zapobiegałoby zapychaniu kolejki wiadomościami, których nikt nie odbierze.
2. **`IPC_NOWAIT` na `msgsnd()`** – nieblokujące wysyłanie. Gdy kolejka jest pełna, proces nie zawiesza się lecz otrzymuje `EAGAIN`, co pozwala na dalsze działanie.
3. **Dynamiczny bufor (`malloc` na starcie)** – przy `EAGAIN` komunikat trafia do bufora w pamięci procesu. Na początku każdej iteracji pętli bufor jest opróżniany do kolejki gdy ta ma wolne miejsce (weryfikowane przez `msgctl(IPC_STAT)`).
4. **`pobierz_stan_kolejki()` (`msgctl IPC_STAT`)** – odczytanie `msg_qnum` (aktualnej liczby wiadomości) pozwala podejmować inteligentne decyzje o próbach odesłania zbuforowanych komunikatów.

F. Obsługa sygnałów: Wezwanie lekarza na oddział (SIGUSR2)

F.1. Konfiguracja handlera w procesie lekarza

```
// lekarz.c - handler i konfiguracja
volatile sig_atomic_t wezwanie_na_oddzial = 0;

void handle_sig(int sig)
{
    if(sig == SIG_LEKARZ_ODDZIAL) wezwanie_na_oddzial = 1;
    else if(sig == SIGINT || sig == SIGTERM) koniec_pracy = 1;
}

// W main():
struct sigaction sa;
sa.sa_handler = handle_sig;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIG_LEKARZ_ODDZIAL, &sa, NULL);  // SIGUSR2
```

Handler jedynie ustawia flagę – właściwa logika wykonuje się w pętli głównej. Każda nazwa procesu specjalisty rozpoczyna się od `SOR_S_*Nazwa_specjalisty*` - jest to pomocne przy egzekwowaniu sygnału z konsoli. handler dla lekarz.c

F.2. Logika opuszczenia SOR przez lekarza

```
// lekarz.c - obsługa w pętli praca_specjalista()
if(wezwanie_na_oddzial) {
    // Wejście do sekcji krytycznej
    while(semop(semid, &lock, 1) == -1) { if(errno!=EINTR) break; }
    stan->dostepni_specjalisci[typ] = 0;  // Oznaczenie jako niedostępny
    while(semop(semid, &unlock, 1) == -1) { if(errno!=EINTR) break; }

    zapisz_raport(KONSOLA, semid, "[%s] Wezwanie na oddzial\n", int_to_lekarz(typ));

    // Symulacja pobytu na oddziale
    sleep(10);

    // Powrót do pracy
    while(semop(semid, &lock, 1) == -1) { if(errno!=EINTR) break; }
    stan->dostepni_specjalisci[typ] = 1;  // Ponowna dostępność
    while(semop(semid, &unlock, 1) == -1) { if(errno!=EINTR) break; }

    wezwanie_na_oddzial = 0;
}
```

Możemy na bieżąco monitorować dyżurujących specjalistów w *spec_na_oddziale.txt*

logika wezwanie na oddział

F.3. Proces Dyrektora (opcjonalny)

Dyrektor jest uruchamiany gdy program `main` otrzyma argument `auto`:

```
// main.c - proces dyrektora
if (argc > 1 && strcmp(argv[1], "auto") == 0) {
    pid_dyrektor = fork();
    if (pid_dyrektor == 0) {
        sleep(1);
        signal(SIGINT, SIG_IGN);   // Ignorowanie SIGINT
        signal(SIGTERM, SIG_IGN);  // Ignorowanie SIGTERM

        StanSOR *stan_child = (StanSOR*)shmat(shmid, NULL, 0);
        srand(time(NULL) ^ getpid());

        while(stan_child->symulacja_trwa) {
            int lek = (rand() % 6) + 1;  // Losowy specjalista 1-6
            if (stan_child->dostepni_specjalisci[lek]) {
                if (pid_lekarze[lek] > 0)
                    kill(pid_lekarze[lek], SIG_LEKARZ_ODDZIAL);
            }
            sleep(rand() % 5 + 2);  // 2-6 sekund przerwy
        }
        exit(0);
    }
}
```

Przy uruchamianiu wpisz w konsoli: *./main auto*

Dyrektor ignoruje sygnały zakończenia, ponieważ sam musi być aktywnie zabity przez `main` podczas procedury ewakuacji (`SIGKILL`).

proces dyrektora

G. Obsługa sygnałów: Procedura ewakuacji (SIGINT)

G.1. Handler w procesie main

handler w main

```
// main.c - handler i flaga
volatile sig_atomic_t ewakuacja_rozpoczeta = 0;

void signal_handler(int sig) {
    if (sig == SIGINT) ewakuacja_rozpoczeta = 1;
}

// Konfiguracja w main():
struct sigaction sa;
sa.sa_handler = signal_handler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGINT, &sa, NULL);
signal(SIGTERM, SIG_IGN);  // Main ignoruje SIGTERM
```

G.2. Funkcja przeprowadz_ewakuacje()

przeprowadz ewakuacje

```
// main.c - procedura ewakuacji
void przeprowadz_ewakuacje() {
    printf("\n=== ROZPOCZYNAM EWAKUACJE SOR ===\n");

    // Wysłanie SIGTERM do personelu medycznego
    if (pid_rejestracja_1 > 0) kill(pid_rejestracja_1, SIGTERM);
    if (pid_poz > 0) kill(pid_poz, SIGTERM);
    for(int i=1; i<=6; i++)
        if(pid_lekarze[i] > 0) kill(pid_lekarze[i], SIGTERM);

    usleep(10000);  // Krótka pauza na obsługę sygnałów

    // Wysłanie SIGINT do generatora (uruchomi procedurę ewakuacji pacjentów)
    if (pid_gen > 0) {
        kill(pid_gen, SIGINT);
        waitpid(pid_gen, NULL, 0);  // Czekanie na zakończenie generatora
    }

    // Zabicie dyrektora (ignoruje sygnały, więc SIGKILL)
    if (pid_dyrektor > 0) kill(pid_dyrektor, SIGKILL);

    // Czekanie na wszystkie procesy potomne
    while(wait(NULL) > 0);
}
```

G.3. Procedura ewakuacji w generatorze

Generator po otrzymaniu SIGINT wykonuje procedurę:

ewakuacja generator

```
// generuj.c - procedura_ewakuacji()
void procedura_ewakuacji() {
    signal(SIGINT, SIG_IGN);
    signal(SIGTERM, SIG_IGN);

    printf("\n[GENERATOR] EWAKUACJA: Blokuje dostep do SHM i robie Snapshot...\n");

    // Zablokowanie dostępu do pamięci dzielonej
    struct sembuf lock = {SEM_DOSTEP_PAMIEC, -1, SEM_UNDO};
    struct sembuf unlock = {SEM_DOSTEP_PAMIEC, 1, SEM_UNDO};
    semop(semid, &lock, 1);

    StanSOR *stan = (StanSOR*)shmat(shmid, NULL, 0);
    if (stan != (void*)-1) {
        // Snapshot stanu przed ewakuacją
        stan->snap_w_srodku = stan->pacjenci_w_poczekalni;
        stan->snap_przed_sor = stan->pacjenci_przed_sor;

        printf("[GENERATOR] Snapshot: W srodku=%d, Przed SOR=%d\n",
               stan->snap_w_srodku, stan->snap_przed_sor);

        // Wysłanie SIGTERM do wszystkich procesów w grupie
        printf("[GENERATOR] Zabijam pacjentow (SIGTERM)...\n");
        kill(0, SIGTERM);

        shmdt(stan);
    }

    semop(semid, &unlock, 1);

    // Zbieranie statusów zakończenia wszystkich dzieci
    int suma_exit_code = 0;
    int status;
    pid_t pid;
    while ((pid = waitpid(-1, &status, 0)) > 0) {
        if (WIFEXITED(status)) {
            suma_exit_code += WEXITSTATUS(status);
        }
    }

    printf("\n[GENERATOR] Ewakuacja zakonczona.\n");
    printf("[GENERATOR] Suma kodow wyjscia (kontrolna): %d\n", suma_exit_code);
}
```

Snapshot umożliwia późniejszą weryfikację, czy liczba ewakuowanych pacjentów zgadza się z liczbą obecnych w systemie.

G.4. Handler ewakuacji w procesie pacjenta

pacjent.c

```
// pacjent.c - handler
void handle_kill(int sig) {
    if (stan_pacjenta == STAN_W_POCZEKALNI) {
        _exit(sem_op_miejsca);  // 1 lub 2 (z opiekunem)
    }
    _exit(0);
}
```

Pacjent używa `_exit()` zamiast `exit()` aby uniknąć wykonywania procedur sprzątających, które mogłyby powodować problemy podczas ewakuacji.

Argumentem _exit() jest wartość sem_op_miejsca - jest to ta sama zmienna która kontroluje o ile wykonujemy V/P na semaforze (rodzic z dzieckiem = 2, dorosły = 1).

H. Wątki monitorujące i raportujące

H.1. Wątek statystyk (watek_statystyki)

Zbiera statystyki z dedykowanej kolejki komunikatów:

watek statystyki: main.c

```
// main.c - wątek statystyk
void* watek_statystyki(void* arg) {
    StatystykaPacjenta msg;
    while (monitor_running) {
        if (msgrcv(msgid_stat, &msg, sizeof(StatystykaPacjenta) - sizeof(long),
                   0, IPC_NOWAIT) == -1) {
            if (errno == ENOMSG || errno == EINTR) { usleep(50000); continue; }
            break;
        }

        pthread_mutex_lock(&stat_mutex);
        statystyki.obs_pacjenci++;
        if (msg.czy_vip) statystyki.ile_vip++;
        if (msg.kolor > 0 && msg.kolor <= 3) statystyki.obs_kolory[msg.kolor]++;
        if (msg.typ_lekarza == 0) statystyki.obs_dom_poz++;
        else if (msg.typ_lekarza >= 1 && msg.typ_lekarza <= 6)
            statystyki.obs_spec[msg.typ_lekarza]++;
        if (msg.skierowanie >= 1 && msg.skierowanie <= 3)
            statystyki.decyzja[msg.skierowanie]++;
        pthread_mutex_unlock(&stat_mutex);
    }

    // Opróżnienie kolejki po zakończeniu symulacji
    while (msgrcv(msgid_stat, &msg, sizeof(StatystykaPacjenta) - sizeof(long),
                  0, IPC_NOWAIT) != -1) {
        // ... aktualizacja statystyk ...
    }
    return NULL;
}
```

Struktura `StatystykiLokalne` jest chroniona mutexem `pthread_mutex_t stat_mutex`, ponieważ może być odczytywana z głównego wątku podczas generowania raportu końcowego.

H.2. Wątek raportu specjalistów (watek_raport_specjalistow)

Okresowo zapisuje status dostępności specjalistów do pliku:

watek dla specjalistow: main.c

```
// main.c - wątek raportu specjalistów
void* watek_raport_specjalistow(void* arg) {
    StanSOR *stan = (StanSOR*)shmat(shmid, NULL, 0);
    if (stan == (void*)-1) return NULL;

    const char* nazwy_spec[] = {"", "Kardiolog", "Neurolog", "Laryngolog",
                                "Chirurg", "Okulista", "Pediatra"};

    while(monitor_running) {
        FILE *f = fopen(RAPORT_2, "w");
        if (f) {
            for(int i=1; i<=6; i++)
                fprintf(f, "%-12s %d\n", nazwy_spec[i], stan->dostepni_specjalisci[i]);
            fclose(f);
        }
        usleep(200000);  // 200ms
    }
    shmdt(stan);
    return NULL;
}
```

Plik `spec_na_oddziale.txt` jest nadpisywany przy każdej iteracji (`"w"` zamiast `"a"`), co zapewnia aktualny stan.

H.3. Zarządzanie cyklem życia wątków

Wszystkie wątki są tworzone na początku i kończone na końcu symulacji:

```
// main.c - tworzenie wątków
pthread_create(&stat_tid, NULL, watek_statystyki, NULL);
pthread_create(&bramka_tid, NULL, watek_bramka, NULL);
pthread_create(&raport2_tid, NULL, watek_raport_specjalistow, NULL);

// main.c - kończenie wątków
if (!ewakuacja_rozpoczeta) {
    monitor_running = 0;  // Sygnalizacja zakończenia
    // ...
}

pthread_join(stat_tid, NULL);
pthread_join(bramka_tid, NULL);
pthread_join(raport2_tid, NULL);
```

Flaga `volatile int monitor_running` kontroluje pętle główne wszystkich wątków. Po jej wyzerowaniu, wątki kończą bieżącą iterację i wychodzą z funkcji.

I. Wątek opiekuna w procesie pacjenta (model zleceniowy)

Zgodnie z wymaganiami projektu, dzieci poniżej 18 lat przychodzą na SOR pod opieką osoby dorosłej. Opiekun jest realizowany jako wątek POSIX w procesie pacjenta. Zamiast pasywnego oczekiwania, opiekun działa w modelu zleceniowym – wątek główny (dziecko) deleguje opiekunowi konkretne zadania IPC, a opiekun je wykonuje i raportuje zakończenie.

I.1. Blok sterujący opiekuna (OpiekunControlBlock)

Synchronizacja między wątkiem dziecka a wątkiem opiekuna opiera się na dedykowanej strukturze z mutexem i zmiennymi warunkowymi:

```
// pacjent.c - typy zadań i blok sterujący
typedef enum {
    BRAK_ZADAN,
    ZADANIE_WEJDZ_SEM,     // Zajmij 1 miejsce w poczekalni (semafor)
    ZADANIE_WYJDZ_SEM,     // Zwolnij 1 miejsce w poczekalni
    ZADANIE_REJESTRACJA,   // Wykonaj komunikację z rejestracją
    ZADANIE_POZ,           // Wykonaj komunikację z POZ
    ZADANIE_SPECJALISTA,   // Wykonaj komunikację ze specjalistą
    ZADANIE_KONIEC         // Zakończ wątek
} TypZadania;

typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t cond_start;   // Opiekun czeka na nowe zadanie
    pthread_cond_t cond_koniec;  // Dziecko czeka na zakończenie zadania
    TypZadania aktualne_zadanie;
    KomunikatPacjenta dane_pacjenta;  // Współdzielone dane komunikatu
} OpiekunControlBlock;

OpiekunControlBlock OpiekunSync;
```

Struktura `OpiekunControlBlock` realizuje wzorzec producent-konsument: dziecko (producent) ustawia zadanie i sygnalizuje `cond_start`, opiekun (konsument) wykonuje zadanie i sygnalizuje `cond_koniec`.

I.2. Funkcja zlecająca zadanie opiekunowi

```
// pacjent.c - zlecanie zadania opiekunowi
void zlec_opiekunowi(TypZadania zadanie) {
    pthread_mutex_lock(&OpiekunSync.mutex);
    OpiekunSync.aktualne_zadanie = zadanie;
    pthread_cond_signal(&OpiekunSync.cond_start);
    while (OpiekunSync.aktualne_zadanie != BRAK_ZADAN) {
        pthread_cond_wait(&OpiekunSync.cond_koniec, &OpiekunSync.mutex);
    }
    pthread_mutex_unlock(&OpiekunSync.mutex);
}
```

Funkcja jest blokująca – dziecko czeka (`cond_wait`) dopóki opiekun nie ustawi `aktualne_zadanie` z powrotem na `BRAK_ZADAN` , co oznacza wykonanie zlecenia.

I.3. Funkcja wątku opiekuna

watek opiekun: pacjent.c

```
// pacjent.c - pętla główna wątku opiekuna
void* watek_opiekun(void* arg) {
    pid_t mpid = getpid();
    pthread_mutex_lock(&OpiekunSync.mutex);
    while (1) {
        while (OpiekunSync.aktualne_zadanie == BRAK_ZADAN)
            pthread_cond_wait(&OpiekunSync.cond_start, &OpiekunSync.mutex);
        if (OpiekunSync.aktualne_zadanie == ZADANIE_KONIEC) break;

        pthread_mutex_unlock(&OpiekunSync.mutex);

        KomunikatPacjenta *msg = &OpiekunSync.dane_pacjenta;
        struct sembuf operacja_sem = {SEM_MIEJSCA_SOR, 0, SEM_UNDO};

        switch (OpiekunSync.aktualne_zadanie) {
            case ZADANIE_WEJDZ_SEM:
                operacja_sem.sem_op = -1;
                while (semop(semid, &operacja_sem, 1) == -1) if(errno!=EINTR) exit(104);
                aktualizuj_liczniki(-1, 1, 0);
                break;
            case ZADANIE_WYJDZ_SEM:
                operacja_sem.sem_op = 1;
                semop(semid, &operacja_sem, 1);
                aktualizuj_liczniki(0, -1, 0);
                break;
            case ZADANIE_REJESTRACJA:
                wykonaj_ipc_samodzielnie(rej_msgid, SLIMIT_REJESTRACJA, msg);
                break;
            case ZADANIE_POZ:
                msg->mtype = 1;
                wykonaj_ipc_samodzielnie(poz_id, SLIMIT_POZ, msg);
                break;
            case ZADANIE_SPECJALISTA:
                if (msg->typ_lekarza > 0) {
                    int spec_id = msg->typ_lekarza;
                    int qid = msgget(ftok(FILE_KEY, (spec_id==1?'K':spec_id==2?'N':spec_id==3?'L':
                                                     spec_id==4?'C':spec_id==5?'O':'D')), 0);
                    msg->mtype = msg->kolor;
                    wykonaj_ipc_samodzielnie(qid, spec_id + 1, msg);
                }
                break;
            default: break;
        }

        pthread_mutex_lock(&OpiekunSync.mutex);
        OpiekunSync.aktualne_zadanie = BRAK_ZADAN;
        pthread_cond_signal(&OpiekunSync.cond_koniec);
    }
    pthread_mutex_unlock(&OpiekunSync.mutex);
    return NULL;
}
```

Opiekun wykorzystuje tę samą funkcję `wykonaj_ipc_samodzielnie()` co dorosły pacjent, dzięki czemu logika komunikacji IPC jest współdzielona. Kluczowa różnica polega na tym, że opiekun operuje na wskaźniku do `OpiekunSync.dane_pacjenta` – współdzielonego bufora komunikatu, przez który dziecko i opiekun wymieniają się danymi.

Wątek opiekuna wykonuje operacje na semaforze z flagą `SEM_UNDO` – oznacza to, że przy awaryjnym zakończeniu procesu, zarówno miejsce dziecka jak i opiekuna zostaną zwolnione niezależnie.

I.4. Tworzenie wątku opiekuna i zajmowanie podwójnego miejsca

```
// pacjent.c - inicjalizacja opiekuna dla nieletnich
int wiek = rand() % 100;

if (wiek < 18) {
    RODZIC_POTRZEBNY = 1;
    sem_op_miejsca = 2;  // Dziecko + opiekun = 2 miejsca w poczekalni
    pthread_mutex_init(&OpiekunSync.mutex, NULL);
    pthread_cond_init(&OpiekunSync.cond_start, NULL);
    pthread_cond_init(&OpiekunSync.cond_koniec, NULL);
    OpiekunSync.aktualne_zadanie = BRAK_ZADAN;
    pthread_create(&rodzic_thread, NULL, watek_opiekun, NULL);

    zapisz_raport(KONSOLA, semid, "[PACJENT %d ] WIEK %d |Z DOROSLYM %lu |\n",
                  mpid, wiek, (unsigned long)rodzic_thread);
} else {
    sem_op_miejsca = 1;
    zapisz_raport(KONSOLA, semid, "[PACJENT  %d ] WIEK %d \n", mpid, wiek);
}
```

Zmienna `sem_op_miejsca` przyjmuje wartość 1 dla dorosłych lub 2 dla nieletnich z opiekunem.

I.5. Przebieg wizyty dziecka – delegacja do opiekuna

Dziecko (wątek główny) zajmuje jedno miejsce w poczekalni bezpośrednio, a drugie miejsce zleca opiekunowi. Następnie każdy etap komunikacji IPC (rejestracja, POZ, specjalista) jest delegowany do opiekuna przez `zlec_opiekunowi()` :

```
// pacjent.c - sekwencja wizyty dziecka z opiekunem
aktualizuj_liczniki(sem_op_miejsca, 0, 0);  // +2 przed SOR

// Opiekun zajmuje swoje 1 miejsce
zlec_opiekunowi(ZADANIE_WEJDZ_SEM);

// Dziecko zajmuje swoje 1 miejsce
struct sembuf wejscie = {SEM_MIEJSCA_SOR, -1, SEM_UNDO};
while (semop(semid, &wejscie, 1) == -1) { if (errno == EINTR) continue; exit(104); }

aktualizuj_liczniki(-1, 1, 1);
stan_pacjenta = STAN_W_POCZEKALNI;

// Rejestracja - delegacja do opiekuna
OpiekunSync.dane_pacjenta = msg;
zlec_opiekunowi(ZADANIE_REJESTRACJA);
msg = OpiekunSync.dane_pacjenta;

aktualizuj_liczniki(0, 0, -1);

// POZ - delegacja do opiekuna
OpiekunSync.dane_pacjenta = msg;
zlec_opiekunowi(ZADANIE_POZ);
msg = OpiekunSync.dane_pacjenta;

// Specjalista - delegacja do opiekuna
if (msg.typ_lekarza > 0) {
    OpiekunSync.dane_pacjenta = msg;
    zlec_opiekunowi(ZADANIE_SPECJALISTA);
    msg = OpiekunSync.dane_pacjenta;
}
```

Po każdym zleceniu dziecko odczytuje zaktualizowane dane z `OpiekunSync.dane_pacjenta` – opiekun mógł np. zmienić pole `kolor` (po triażu) czy `skierowanie` (po specjaliście).

I.6. Kończenie wątku opiekuna

```
// pacjent.c - zakończenie normalne
if (RODZIC_POTRZEBNY)
{
    // Zwolnienie miejsca opiekuna w poczekalni
    zlec_opiekunowi(ZADANIE_WYJDZ_SEM);

    // Zakończenie wątku opiekuna
    pthread_mutex_lock(&OpiekunSync.mutex);
    OpiekunSync.aktualne_zadanie = ZADANIE_KONIEC;
    pthread_cond_signal(&OpiekunSync.cond_start);
    pthread_mutex_unlock(&OpiekunSync.mutex);
    pthread_join(rodzic_thread, NULL);
}

// Dziecko zwalnia swoje miejsce
struct sembuf wyjscie = {SEM_MIEJSCA_SOR, 1, SEM_UNDO};
semop(semid, &wyjscie, 1);
aktualizuj_liczniki(0, -1, 0);
```

Zadanie `ZADANIE_KONIEC` powoduje wyjście opiekuna z pętli głównej. `pthread_join()` czeka na faktyczne zakończenie wątku i zwalnia jego zasoby.

J. System stanów pacjenta

J.1. Definicje stanów

```
// wspolne.h - stany pacjenta
#define STAN_PRZED_SOR 0      // Czeka na wejście
#define STAN_W_POCZEKALNI 1   // Wewnątrz SOR
#define STAN_WYCHODZI 2       // Zakończona obsługa
```

J.2. Przejścia między stanami

stany pacjenta: pacjent.c

```
// pacjent.c - sekwencja stanów
stan_pacjenta = STAN_PRZED_SOR;

// Aktualizacja licznika osób przed SOR
aktualizuj_liczniki(sem_op_miejsca, 0, 0);

// Oczekiwanie na miejsce w poczekalni (semafor)
struct sembuf wejscie = {SEM_MIEJSCA_SOR, -sem_op_miejsca, SEM_UNDO};
while (semop(semid, &wejscie, 1) == -1) { /* ... */ }

// Wejście do poczekalni
aktualizuj_liczniki(-sem_op_miejsca, sem_op_miejsca, 1);
stan_pacjenta = STAN_W_POCZEKALNI;

// ... obsługa przez rejestrację, POZ, specjalistę ...

// Wyjście z SOR
stan_pacjenta = STAN_WYCHODZI;
aktualizuj_liczniki(0, -sem_op_miejsca, 0);
```

Stan pacjenta jest wykorzystywany w handlerze ewakuacji do określenia, jakie zasoby należy zwolnić 0 / sem_op_miejsca.

K. Funkcje pomocnicze

K.1. Funkcja zapisz_raport()

zapisz_raport: wspolne.h

```
// wspolne.h - funkcja raportująca
static inline void zapisz_raport(const char* filename, int semid, const char* format, ...) {
    (void)semid;  // Parametr zachowany dla kompatybilności
    char bufor[1024];
    va_list args;
    va_start(args, format);
    int len = vsnprintf(bufor, sizeof(bufor), format, args);
    va_end(args);
    if (len <= 0) return;

    if (filename == KONSOLA) {
        write(STDOUT_FILENO, bufor, len);
    } else {
        int fd = open(filename, O_WRONLY | O_CREAT | O_APPEND, 0600);
        if (fd != -1) {
            write(fd, bufor, len);
            close(fd);
        }
    }
}
```

Funkcja używa `write()` zamiast `printf()` / `fprintf()`, co jest bezpieczniejsze w kontekście signal-handlerów. Pierwszym argumentem funkcji jest wyjście - plik raportowy, bądź konsola, semid - pozostałość po starej implementacji.

K.2. Funkcja podsumowanie()

Generuje raport końcowy porównujący obserwowane statystyki z wartościami oczekiwanymi:

funkcja ostatecznego podsumowania: wspolne.h

```
// wspolne.h - funkcja podsumowująca
static inline void podsumowanie(StatystykiLokalne *stat, StanSOR *stan)
{
    double p = (double)stat->obs_pacjenci;
    if (p == 0) p = 1.0;

    int ewak_z_poczekalni = stan->snap_w_srodku;
    int ewak_sprzed_sor = stan->snap_przed_sor;

    // ... formatowanie i wyświetlanie statystyk ...

    pos += sprintf(bufor + pos, "Pacjenci VIP:    %d (oczekiwano ok.: %d)\n",
                   stat->ile_vip, (int)(0.2 * p + 0.5));
    pos += sprintf(bufor + pos, "Czerwony: %d (oczekiwano ok.: %d)\n",
                   stat->obs_kolory[CZERWONY], (int)(0.1 * p + 0.5));
    // ... itd. ...

    write(STDOUT_FILENO, bufor, pos);
}
```

K.3. Funkcja uruchom_proces()

Wrapper standaryzujący tworzenie procesów potomnych:

tworzenie procesów potomnych - funkcja pomocnicza: main.c

```
// main.c - wrapper uruchamiania procesów
pid_t uruchom_proces(const char* prog, const char* name, const char* arg1) {
    pid_t pid = fork();
    if (pid == 0) {
        signal(SIGINT, SIG_DFL);   // Przywrócenie domyślnej obsługi
        signal(SIGTERM, SIG_DFL);
        if(arg1) execl(prog, name, arg1, NULL);
        else execl(prog, name, NULL);
        exit(1);  // Tylko jeśli execl() zawiedzie
    }
    return pid;
}
```

Przywrócenie domyślnej obsługi sygnałów w procesie potomnym jest kluczowe, ponieważ dyspozycje sygnałów są dziedziczone po `fork()`.

K.4. Funkcja czyszczenie()

czyszczenie: main.c

Zwalnia wszystkie zasoby IPC:

```
// main.c - procedura sprzątająca
void czyszczenie() {
    if (shmid != -1) shmctl(shmid, IPC_RMID, NULL);
    if (semid != -1) semctl(semid, 0, IPC_RMID);
    if (semid_limits != -1) semctl(semid_limits, 0, IPC_RMID);
    for (int i = 0; i < 20; i++)
        if (msgs_ids[i] != -1) msgctl(msgs_ids[i], IPC_RMID, NULL);
    if (msgid_stat != -1) msgctl(msgid_stat, IPC_RMID, NULL);
    printf("\n[SYSTEM] Wykonano czyszczenie zasobow IPC.\n");
}
```

Flaga `IPC_RMID` oznacza natychmiastowe usunięcie zasobu. Dla pamięci dzielonej, faktyczne zwolnienie nastąpi gdy ostatni proces odłączy się od segmentu.

Podręcznik do semaforów

Zbiór (klucz)	Indeks	Nazwa	Wartość Początkowa	Opis działania
ID_SEM_SET ('M')	0	SEM_DOSTEP_PAMIEC	1	Mutex chroniący strukturę StanSOR przed jednoczesnym zapisem (race conditions).
ID_SEM_SET ('M')	1	SEM_MIEJSCA_SOR	MAX_PACJENTOW	Licznik miejsc w poczekalni. Blokuje wejście przy pełnym obłożeniu.
ID_SEM_SET ('M')	2	SEM_ZAPIS_PLIK	1	Mutex I/O (obecnie nieużywany - zachowany dla kompatybilności).
ID_SEM_SET ('M')	3	SEM_GENERATOR	MAX_PROCESOW	Limit jednoczesnych procesów pacjentów. Chroni przed fork-bombą.
ID_SEM_LIMITS ('X')	0	SLIMIT_REJESTRACJA	INT_LIMIT_KOLEJEK	Limit komunikatów w kolejce rejestracji.
ID_SEM_LIMITS ('X')	1	SLIMIT_POZ	INT_LIMIT_KOLEJEK	Limit komunikatów w kolejce POZ.
ID_SEM_LIMITS ('X')	2-7	SLIMIT_[SPECJALISTA]	INT_LIMIT_KOLEJEK	Limity dla kolejek specjalistów (Kardiolog, Neurolog, Laryngolog, Chirurg, Okulista, Pediatra).

Podręcznik do kolejek komunikatów

Klucz	Identyfikator	Przeznaczenie	Typy wiadomości
'R'	ID_KOLEJKA_REJESTRACJA	Komunikacja pacjent ↔ rejestracja	mtype=1 (VIP), mtype=2 (Zwykły), mtype=PID (odpowiedź)
'P'	ID_KOLEJKA_POZ	Komunikacja pacjent ↔ lekarz POZ	mtype=1 (do POZ), mtype=PID (odpowiedź)
'K'	ID_KOL_KARDIOLOG	Komunikacja pacjent ↔ kardiolog	mtype=1,2,3 (kolor), mtype=PID (odpowiedź)
'N'	ID_KOL_NEUROLOG	Komunikacja pacjent ↔ neurolog	j.w.
'L'	ID_KOL_LARYNGOLOG	Komunikacja pacjent ↔ laryngolog	j.w.
'C'	ID_KOL_CHIRURG	Komunikacja pacjent ↔ chirurg	j.w.
'O'	ID_KOL_OKULISTA	Komunikacja pacjent ↔ okulista	j.w.
'D'	ID_KOL_PEDIATRA	Komunikacja pacjent ↔ pediatra	j.w.
'T'	ID_KOLEJKA_STATYSTYKI	Zbieranie statystyk przez wątek	mtype=1 (StatystykaPacjenta)

Stałe konfiguracyjne

Stała	Wartość / maksymalna przepustowość	Opis
PACJENCI_NA_DOBE	max zakres int	Całkowita liczba pacjentów do wygenerowania
MAX_PACJENTOW	< 32767	Maksymalna pojemność poczekalni SOR
MAX_PROCESOW	< 32767	Maksymalna liczba jednoczesnych procesów pacjentów
INT_LIMIT_KOLEJEK	< 628	Maksymalna liczba komunikatów w każdej kolejce
PROG_OTWARCIA	MAX_PACJENTOW/2	Próg otwarcia drugiego okienka
PROG_ZAMKNIECIA	MAX_PACJENTOW/3	Próg zamknięcia drugiego okienka

Kompilacja i uruchomienie

Wymagania

- **Kompilator:** gcc
- **System operacyjny:** Linux
- **Biblioteki:** pthread, standardowe biblioteki systemowe IPC

Kompilacja

Projekt wykorzystuje `Makefile` do kompilacji. Wszystkie pliki źródłowe są kompilowane z flagami `-Wall` (wszystkie ostrzeżenia) oraz `-pthread` (dla wątków).

```
 # Kompilacja wszystkich programów
make all

# Lub po prostu:
make
```

Makefile tworzy następujące pliki wykonywalne:

- `main` – główny proces zarządzający symulacją
- `pacjent` – program procesu pacjenta

- `lekarz` – program procesu lekarza (POZ i specjaliści)
- `rejestracja` – program procesu rejestracji
- `generator` – generator procesów pacjentów

Uruchomienie

Tryb podstawowy (bez automatycznego dyrektora)

```
./main
```

W tym trybie symulacja działa bez automatycznych wezwań lekarzy na oddział. Wezwania można wysyłać ręcznie z konsoli za pomocą polecenia `kill`:

```
# Wysłanie sygnału SIGUSR2 do konkretnego lekarza (wymaga znajomości PID)
kill -SIGUSR2
```

Tryb automatyczny (z procesem dyrektora)

```
./main auto
```

W tym trybie uruchamiany jest dodatkowy proces dyrektora, który losowo wybiera specjalistów i wysyła im sygnały wezwania na oddział (`SIGUSR2`). Interwał między wezwaniami wynosi 2-6 sekund.

Zatrzymanie symulacji

Normalne zakończenie

Symulacja kończy się automatycznie po wygenerowaniu i obsłużeniu wszystkich pacjentów (`PACJENCI_NA_DOBE`).

Ewakuacja (przerwanie)

```
# Naciśnij Ctrl+C w terminalu z uruchomionym ./main
# lub wyślij sygnał SIGINT:
kill -SIGINT
```

Wywołuje to procedurę ewakuacji – wszystkie procesy są bezpiecznie zamykane, a na końcu wyświetlany jest raport z danymi o ewakuowanych pacjentach.

Czyszczenie

```
# Usunięcie plików wykonywalnych i raportów
make clean

# Usunięcie zasobów IPC (w przypadku niespodziewanego zakończenia)
make ipc_clean
```

Testy

T1: Czy Użycie semafora pred kolejką komunikatów skutecznie uniemożliwia zapchanie kolejki i w konsekwencji chroni przed deadlockiem (exit pacjenta w sekcji krytycznej)?

Będe używał stwierdzenia procesy SOR - chodzi mi o rejestracje, POZ, specjalistów - implementacja komunikacji pomiędzy pacjentem, a każdym procesem SOR jest taka sama.

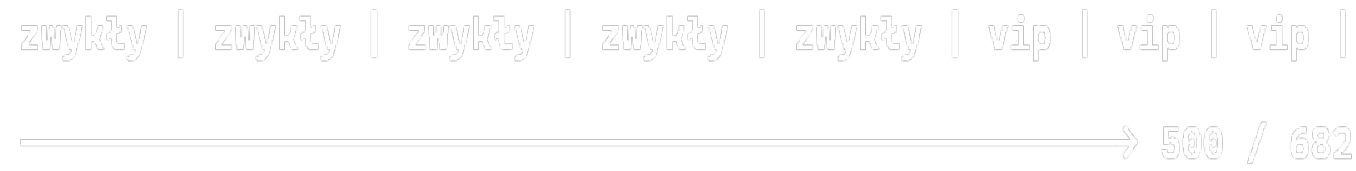
Założena:

- Każda z 8 kolejek komunikatów posiada osobisty semafor obniżany w przed wysłaniem wiadomości do procesu SOR, podwyższany po otrzymaniu wiadomości zwrotnej.
- Każda kolejka ma maksymalnie 682 slotów (systemowy limit w bajtach / sizeof(msg) - sizeof(long)) - pozostałe procesy czekają. Wartość semaforów musi być < 682.
- Każdy proces SOR przyjmuję według ustalonego wcześniej priorytetu - Pacjenci VIP / Pacjenci z kolorem czerwonym powinni być obsłużeni szybciej od pozostałych.

Potencjalne problemy:

- Każdy Pacjent wykonuje operację semop z flagą `SEM_UNDO` - jest ona newralgiczna z bardzo prostego powodu - kiedy Pacjent zexituje / dostanie sygnał terminujący w sekcji krytycznej (przed podwyższeniem semafora, który wcześniej obniżał) System automatycznie wykona inkrementację na semaforze o wartości op, którą proces użył "wchodząc" przez semafor - nie blokując tym samym semafora do końca wykonywania programu.
- Użycie `SEM_UNDO` jest wskazane, ale budzi ono szereg innych problemów. Jeśli semafor jest jedyną blokadą przed faktyczną drogą komunikacji istnieje możliwość, że pacjent po wysłaniu wiadomości do processu SOR, a jeszcze przed otrzymaniem wiadomości zwrotnej dostanie sygnał terminujący jego działanie (w testach zwykły exit wystarczy). System automatycznie dokona inkrementacji na semaforze i w konsekwencji przez semafor wejdzie kolejny pacjent. Oczywiście jeden exitujacy proces nie spowoduje wiele szkód, ale co jeśli takich pacjentów jest 500? Nagle Kolejka jest 1000 wiadomości, a proces SOR - np. rejestracja nie ma wolnych slotów do wysłania wiadomości zwrotnej - mamy deadlocka.
- Od razu nasuwa się pomysł żeby po msgrcv proces SOR patzrył czy proces w ogóle istnieje - znamy jego PID z wiadomośći, którą przesyła oraz posiadamy permisję - wystarczy kill(pid, 0). Jednak to tylko częściowo rozwiąże nasz problem z prostego powodu:

Tak wygląda nasza kolejka gdy pierwsze 500 pacjentów obniża semafor i wysyła wiadomosc i załóżmy, że robi sleep(5) - nie robi msgrcv:



Mamy FIFO posortowane według priorytetu, gdzie vip o wartość | -1 | będzie obsłużony szybciej od zwykłego | -2 |

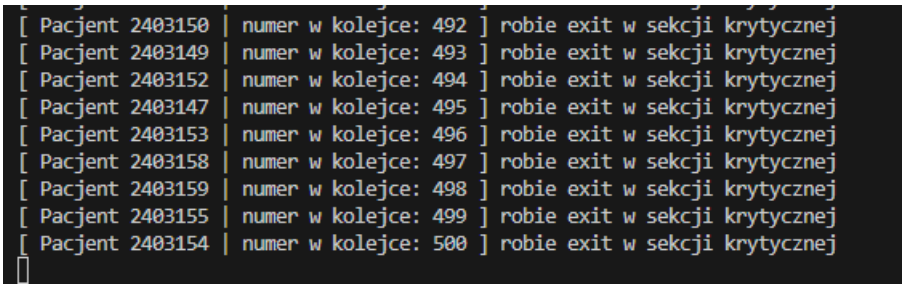
W przypadku wykonania `SEM_UNDO` przez semafor wchodzi kolejne 500 osób, ich wiadomości są wymieszane z tymi, którzy już nie żyją, ALE co najważniejsze - posortowane według priorytetu.

szary kolor oznacza nieżywych pacjentów*

Proces SOR zobaczy, że pierwsze 3 pacjentów już nie istnieje - skutecznie opróżni kolejke o te (3) wiadomości, jednak następnie trafi na "żywego" vip i się zablokuje bo l. wiadomości nadal jest > 682 (limit).

- Rozwiązaniem jest dynamiczny bufor, który - gdy kolejka jest pełna - zapisze strukture żywego pacjenta, następnie przejdzie do anulowania kolejnego nieżywego pacjenta.

Przebieg testu:

- rejestracja wykonuje sleep(60) - czeka na zapchanie kolejki
- wchodzi pierwsze 500 pacjentow - obnizaja semafor, robia exit(0) pomiedzy msgsnd, a msgrcv.
- 

- w ipcs -q nadal widzimy 500 wiadomości - wiadomości istnieją - pacjenci nie.
- wchodzi kolejne 500 pacjentów - zapełniają kolejkę do maksymalnego limitu.
- rejestracja się budzi i dokonuje czyszczenia

```
[ Rejestracja 1 ] Brak informacji o pacjencie 2402738 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402740 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] KOLEJKA FULL (682/682 msg) Pacjent 2403833 -> BUFOR (82/682).
[ Rejestracja 1 ] Brak informacji o pacjencie 2402734 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402744 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402746 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402747 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] KOLEJKA FULL (682/682 msg) Pacjent 2403486 -> BUFOR (83/682).
[ Rejestracja 1 ] Brak informacji o pacjencie 2402748 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402750 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402754 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402753 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402755 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402751 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402752 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402760 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402757 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402762 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] KOLEJKA FULL (682/682 msg) Pacjent 2403585 -> BUFOR (84/682).
[ Rejestracja 1 ] Pacjent 2403541 -> POZ
[ Rejestracja 1 ] Brak informacji o pacjencie 2402759 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Pacjent 2403854 -> POZ
[ Rejestracja 1 ] Pacjent 2403514 -> POZ
[ Rejestracja 1 ] Pacjent 2403579 -> POZ
[ Rejestracja 1 ] Brak informacji o pacjencie 2402761 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Wznowiono z bufora: 2403168
[ Rejestracja 1 ] Brak informacji o pacjencie 2402765 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Wznowiono z bufora: 2403174
[ Rejestracja 1 ] Pacjent 2403867 -> POZ
[ Rejestracja 1 ] Brak informacji o pacjencie 2402771 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Brak informacji o pacjencie 2402768 (exit), Anuluje wysylanie wiadomosci
[ Rejestracja 1 ] Wznowiono z bufora: 2403190
```

Podgląd w IPCS:

```
utrata.hugo.155238@torus:~/sor_hu$ ipcs -q | grep 0x520009b3
0x520009b3 15761445   utrata.hug 600        12000        500
utrata.hugo.155238@torus:~/sor_hu$ ipcs -q | grep 0x520009b3
0x520009b3 15761445   utrata.hug 600        12000        500
utrata.hugo.155238@torus:~/sor_hu$ ipcs -q | grep 0x520009b3
0x520009b3 15761445   utrata.hug 600        16368        682
utrata.hugo.155238@torus:~/sor_hu$ ipcs -q | grep 0x520009b3
0x520009b3 15761445   utrata.hug 600        0            0
utrata.hugo.155238@torus:~/sor_hu$
```

T2: Użycie sygnału SIG_LEKARZ_ODDZIAŁ na wszystkich specjalistach równocześnie

Modyfikacje:

- Przed rozpoczęciem pętli głownej - każdy specjalisty robi:

```
raise(SIG_LEKARZ_ODDZIAL);

   while(!koniec_pracy)
   {
       ...
   }
```

- analogicznym byłoby użycie komendy: pkill -SIGUSR2 -f SOR_S_
- na potrzeby testu lekarz specjalista śpi dokładnie 10 sekund

Przebieg testu:

- Stan przed:

spec_na_oddziale.txt

```
1   Kardiolog    1
2   Neurolog     1
3   Laryngolog   1
4   Chirurg      1
5   Okulista     1
6   Pediatra     1
7
```

- Stan w trakcie sleepa lekarzy:

spec_na_oddziale.txt

```
Kardiolog    0
Neurolog     0
Laryngolog   0
Chirurg      0
Okulista     0
Pediatra     0
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

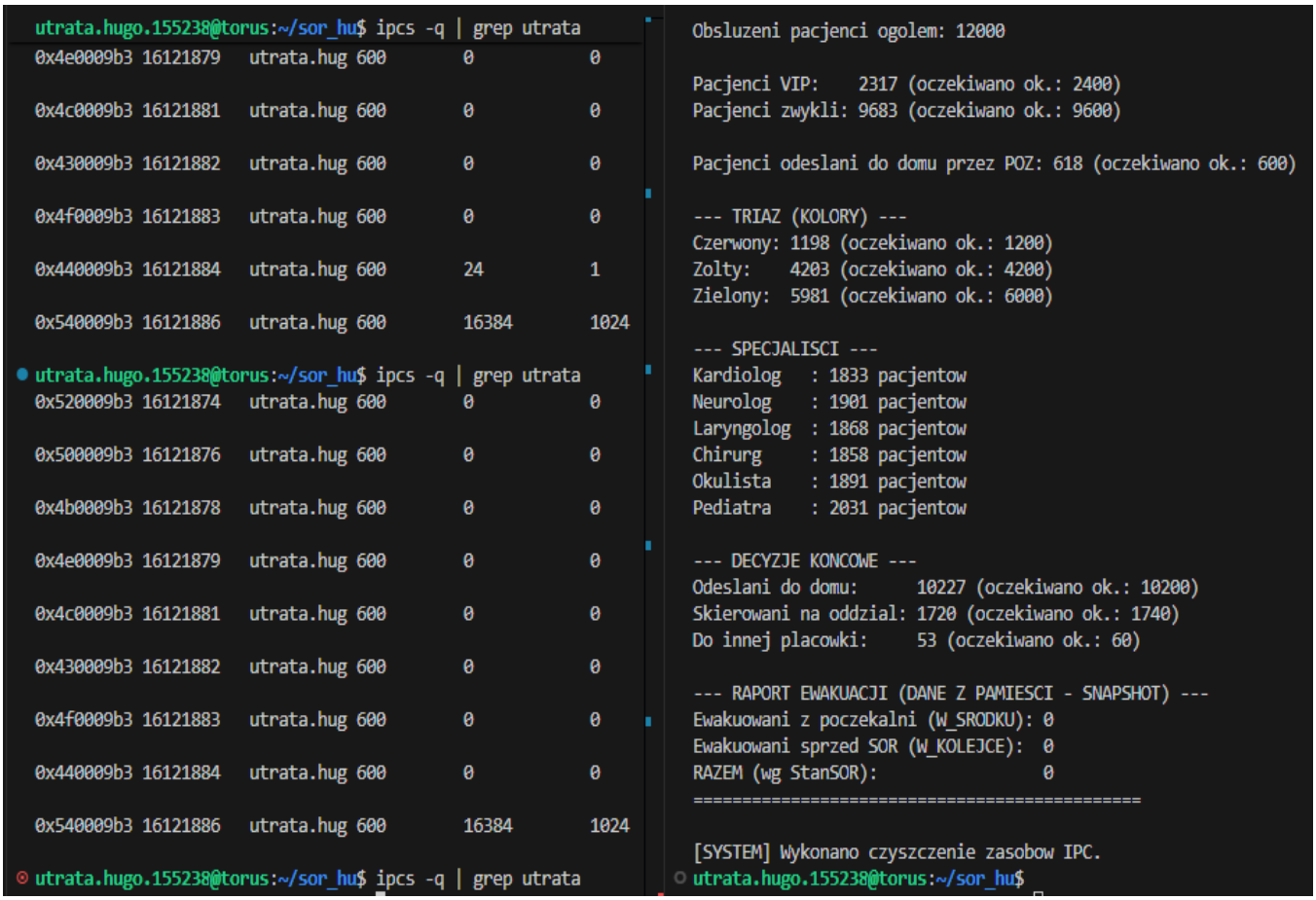
```
utrata.hugo.155238@torus:~/sor_hu$ ipcs -q | grep utrata
0x520009b3 16121874   utrata.hug 600        0            0
0x500009b3 16121876   utrata.hug 600        0            0
0x4b0009b3 16121878   utrata.hug 600        12000        500
0x4e0009b3 16121879   utrata.hug 600        12000        500
0x4c0009b3 16121881   utrata.hug 600        12000        500
0x430009b3 16121882   utrata.hug 600        12000        500
0x4f0009b3 16121883   utrata.hug 600        12000        500
0x440009b3 16121884   utrata.hug 600        12000        500
0x540009b3 16121886   utrata.hug 600        0            0
utrata.hugo.155238@torus:~/sor_hu$
```

- W tym momencie ostatnia kolejka: pacjent -> main jest mocno przeciążona - w pewnym momencie każdy pacjent już skończył swój cykl i nie będziemy dostawać logów na konsoli - jest to OK.

```
utrata.hugo.155238@torus:~/sor_hu$ ipcs -q | grep utrata
0x520009b3 16121874   utrata.hug 600        0            0
0x500009b3 16121876   utrata.hug 600        0            0
0x4b0009b3 16121878   utrata.hug 600        0            0
0x4e0009b3 16121879   utrata.hug 600        0            0
0x4c0009b3 16121881   utrata.hug 600        0            0
0x430009b3 16121882   utrata.hug 600        0            0
0x4f0009b3 16121883   utrata.hug 600        0            0
0x440009b3 16121884   utrata.hug 600        0            0
0x540009b3 16121886   utrata.hug 600        16384        1024
utrata.hugo.155238@torus:~/sor_hu$
```

```
[ Rejestracja 1 ] Pacjent 2247890 -> POZ
[ Rejestracja 1 ] Pacjent 2248558 -> POZ
[ POZ ] Pacjent 2247629 -> Pediatra (Kolor: 2)
[ POZ ] Pacjent 2248996 -> Pediatra (Kolor: 3)
[ POZ ] Pacjent 2248558 -> Pediatra (Kolor: 3)
[ POZ ] Pacjent 2247890 -> Pediatra (Kolor: 2)
[ Pediatra ] Badanie pacjenta 2247629 -> do domu
[ Pediatra ] Badanie pacjenta 2247890 -> do domu
[ Pediatra ] Badanie pacjenta 2248558 -> do domu
[ Pediatra ] Badanie pacjenta 2248996 -> na inny oddział
[ Rejestracja 1 ] Pacjent 2249053 -> POZ
[ POZ ] Pacjent 2249053 -> Pediatra (Kolor: 3)
[ Pediatra ] Badanie pacjenta 2249053 -> do domu
[ Rejestracja 1 ] Pacjent 2249432 -> POZ
[ Rejestracja 1 ] Pacjent 2249660 -> POZ
[ POZ ] Pacjent 2249432 -> Pediatra (Kolor: 3)
[ POZ ] Pacjent 2249660 -> Pediatra (Kolor: 3)
[ Pediatra ] Badanie pacjenta 2249432 -> do domu
[ Pediatra ] Badanie pacjenta 2249660 -> do domu
```

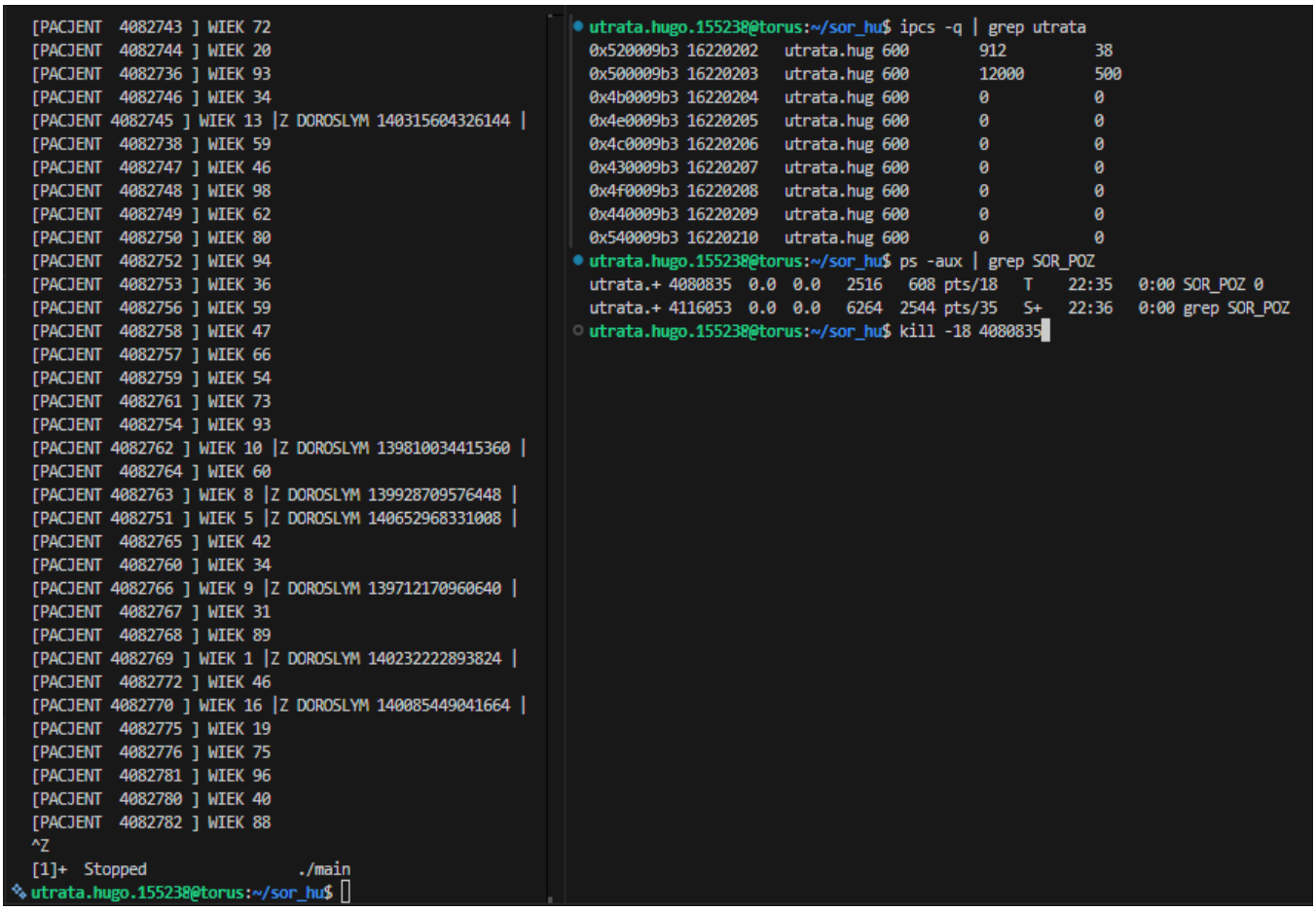
- Po ok. minucie program kończy się sukcesem - kolejki zostały usunięte



T3: (Poprawiony test) SIGTSTP, SIGCONT i działanie kolejki

Przebieg:

- W lekarz POZ robimy sleep(5); - chcemy zebrac jak najwięcej wiadomości od pacjentów
- wykonujemy CTRL + Z (SIGTSTP) do wszystkich procesów



- wykonujemy SIGCONT na samym lekarzu POZ
- POZ się budzi i i wykona 500 razy pętle msgrcv msgsnd po czym się zatrzyma

```
[ POZ ] Pacjent 4081406 -> Okulista (Kolor: 3)
[ POZ ] Pacjent 4081400 -> Laryngolog (Kolor: 2)
[ POZ ] Pacjent 4081408 -> Kardiolog (Kolor: 3)
[ POZ ] Pacjent 4081396 -> Pediatra (Kolor: 2)
[ POZ ] Pacjent 4081413 -> Laryngolog (Kolor: 2)
[ POZ ] Pacjent 4081414 -> Okulista (Kolor: 3)
[ POZ ] Pacjent 4081404 -> Pediatra (Kolor: 1)
[ POZ ] Pacjent 4081415 -> Okulista (Kolor: 3)
[ POZ ] Pacjent 4081418 -> Neurolog (Kolor: 1)
[ POZ ] Pacjent 4081405 -> Kardiolog (Kolor: 2)
[ POZ ] Pacjent 4081419 -> Laryngolog (Kolor: 3)
[ POZ ] Pacjent 4081417 -> Laryngolog (Kolor: 3)
[ POZ ] Pacjent 4081424 -> Laryngolog (Kolor: 2)
[ POZ ] Pacjent 4081428 -> Neurolog (Kolor: 2)
[ POZ ] Pacjent 4081429 -> Okulista (Kolor: 3)
[ POZ ] Pacjent 4081431 -> Chirurg (Kolor: 2)
[ POZ ] Pacjent 4081430 -> Kardiolog (Kolor: 2)
[ POZ ] Pacjent 4081437 -> Okulista (Kolor: 2)
[ POZ ] Pacjent 4081422 -> Pediatra (Kolor: 2)
[ POZ ] Pacjent 4081445 -> Neurolog (Kolor: 1)
[ POZ ] Pacjent 4081425 -> Chirurg (Kolor: 3)
[ POZ ] Pacjent 4081421 -> Pediatra (Kolor: 3)
[ POZ ] Pacjent 4081443 -> Chirurg (Kolor: 3)
[ POZ ] Pacjent 4081446 -> Laryngolog (Kolor: 2)
[ POZ ] Pacjent 4081444 -> Neurolog (Kolor: 2)
[ POZ ] Pacjent 4081449 -> do domu
[ POZ ] Pacjent 4081447 -> Neurolog (Kolor: 2)
[ POZ ] Pacjent 4081452 -> Okulista (Kolor: 3)
[ POZ ] Pacjent 4081455 -> Chirurg (Kolor: 2)
[ POZ ] Pacjent 4081454 -> Okulista (Kolor: 3)
[ POZ ] Pacjent 4081458 -> Chirurg (Kolor: 2)
[ POZ ] Pacjent 4081456 -> Laryngolog (Kolor: 3)
[ POZ ] Pacjent 4081462 -> Kardiolog (Kolor: 3)
[ POZ ] Pacjent 4081435 -> Pediatra (Kolor: 2)
[ POZ ] Pacjent 4081468 -> Neurolog (Kolor: 2)
[ POZ ] Pacjent 4081477 -> Laryngolog (Kolor: 1)
[ POZ ] Pacjent 4081486 -> Kardiolog (Kolor: 3)
```

```
utrata.hugo.155238@torus:~/sor_hu$ ipcs -q | grep utrata
0x520009b3 16220202   utrata.hug 600        912          38
0x500009b3 16220203   utrata.hug 600        12000        500
0x4b0009b3 16220204   utrata.hug 600        0            0
0x4e0009b3 16220205   utrata.hug 600        0            0
0x4c0009b3 16220206   utrata.hug 600        0            0
0x430009b3 16220207   utrata.hug 600        0            0
0x4f0009b3 16220208   utrata.hug 600        0            0
0x440009b3 16220209   utrata.hug 600        0            0
0x540009b3 16220210   utrata.hug 600        0            0
utrata.hugo.155238@torus:~/sor_hu$ ps -aux | grep SOR_POZ
utrata.+ 4080835  0.0  0.0   2516   608 pts/18   T    22:35   0:00 SOR_POZ 0
utrata.+ 4116053  0.0  0.0   6264  2544 pts/35   S+   22:36   0:00 grep SOR_POZ
utrata.hugo.155238@torus:~/sor_hu$ kill -18 4080835
utrata.hugo.155238@torus:~/sor_hu$ ps -aux | grep SOR_POZ
utrata.+  117077  0.0  0.0   6264  2496 pts/35   S+   22:37   0:00 grep SOR_POZ
utrata.+ 4080835  0.0  0.0   2516   608 pts/18   S    22:35   0:00 SOR_POZ 0
utrata.hugo.155238@torus:~/sor_hu$ ipcs -q | grep utrata
0x520009b3 16220202   utrata.hug 600        912          38
0x500009b3 16220203   utrata.hug 600        12000        500
0x4b0009b3 16220204   utrata.hug 600        0            0
0x4e0009b3 16220205   utrata.hug 600        0            0
0x4c0009b3 16220206   utrata.hug 600        0            0
0x430009b3 16220207   utrata.hug 600        0            0
0x4f0009b3 16220208   utrata.hug 600        0            0
0x440009b3 16220209   utrata.hug 600        0            0
0x540009b3 16220210   utrata.hug 600        0            0
utrata.hugo.155238@torus:~/sor_hu$
```

- Liczba wiadomośći nadal wynosi 500 - ale są to wiadomości do odebrania przez pacjentów
- Po wypisaniu 500 logów konsola stoi - dopóki nie wznowimy wszystkich procesów przez SIGCONT

Co poszło nie tak w poprzednim teście?

Poprzednia implementacja:

```
        if(msgrcv(msgid_poz, &pacjent, rozmiar_msg, 0, IPC_NOWAIT) == -1)
        {
            if (errno == ENOMSG || errno == EINTR) { usleep(50000); continue; }
            break;
        }
```

Obecna:

```
        if(msgrcv(msgid_poz, &pacjent, rozmiar_msg, -1, IPC_NOWAIT) == -1)
        {
            if (errno == ENOMSG || errno == EINTR) { usleep(50000); continue; }
            break;
        }
```

Jak widać zmieniliśmy jedynie argument msgtyp (4) - W projekcie założyliśmy, że lekarz POZ przyjmuje bez priorytetów, potraktowałem to jako "Bierz pierwszą lepszą wiadomość" msgtyp = 0 zakłada właśnie taki scenariusz - jednak w przypadku kiedy POZ nie dostaje nowych wiadomości, a stare nie są odbierane przez pacjentów - zaczyna przetwarzać te same wiadomości, które przed chwilą sam wysłał. Skutkiem czego jest niekończoncząca się pętla tych samych 500 pacjentów. Lekcja z tego jest następująca: nawet jeśli nie zależy nam na konkretnym priotytecie - wiadomość musi mieć jakis stały mtype dla wszystkich pacjentów oraz po stronie msgrcv musimy ustawić jakikolwiek msgtyp z tego przedziału żeby odróżnić ją od wiadomości zwrotnej z mtype = pid_pacjenta. Np. "-1" - msgrcv weźmie najmniejszą liczbę mniejszą niż | -1 |.

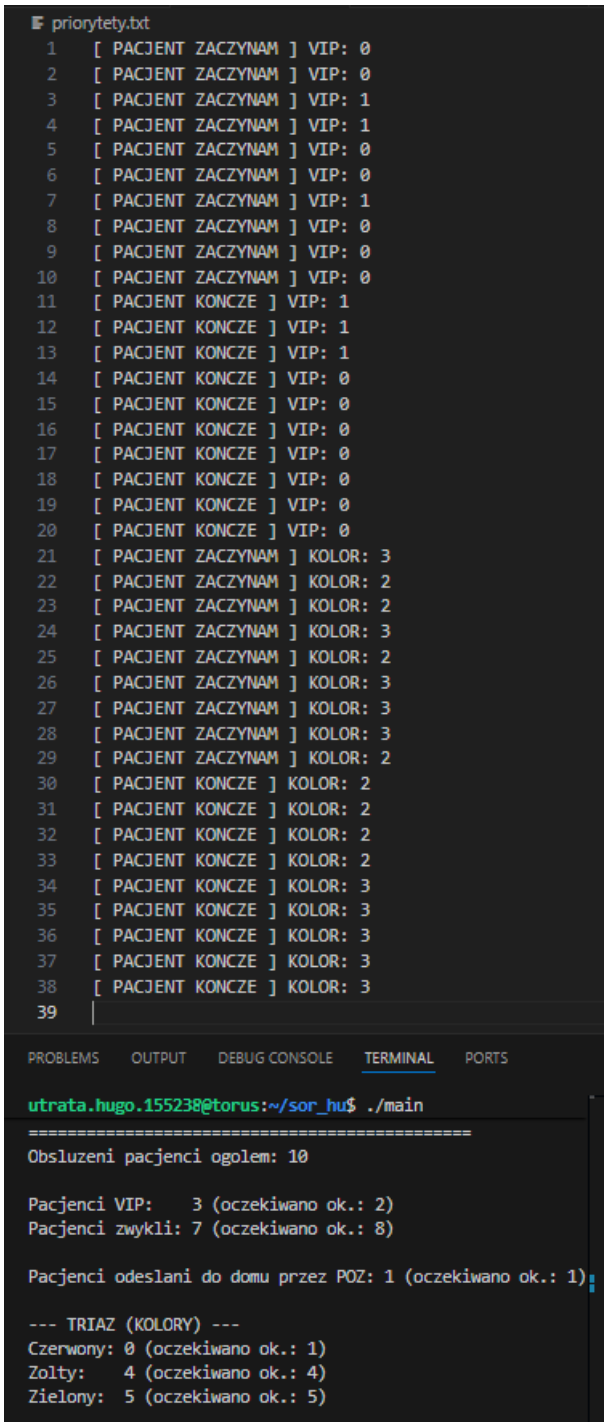
T4: Czy pacjenci VIP, z kolerem Czerownym zostaną obsłużeni szybciej od reszty?

typ vip = 1, typ zwykly = 0; typ czerwony = 1, typ zółty = 2, typ zielony = 3;

Modyfikacje:

- Po każdej etapie komunikacji pacjent robi sleep(3);
- logujemy w pliku priorytety.txt

Wyniki:



Zgodnie z założeniami Pacjenci o niższej wartości mtype (wyższym priorytecie) zostali obsłużeni szybciej

T5: Dynamiczna Bramka nr 2

- **Konfiguracja testowa:** MAX_PACJENTOW = 800, PACJENCI_NA_DOBE = 50000
- **Próg otwarcia:** >= 400 osób
- **Próg zamknięcia:** < 266 osób

Analiza pliku `monitor_bramek.txt` pokazuje poprawne działanie mechanizmu monitorującego otwarcia bramek:

monitor_bramek.txt

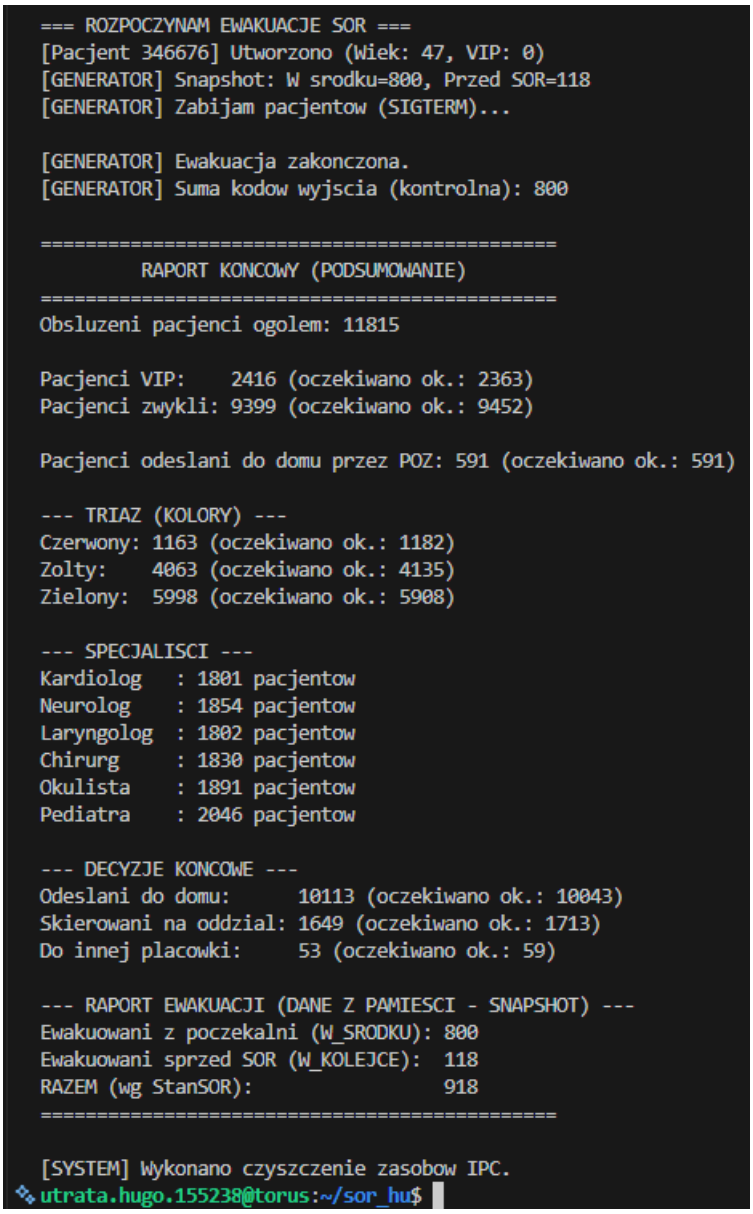
```
[PACJENT] Zlecam OTWARCIE (Kolejka: 400 >= 400)
[MONITOR] Otwieram bramke nr 2
[PACJENT] Zlecam ZAMKNIECIE (Kolejka: 265 < 266)
[MONITOR] Zamykam bramke nr 2
[PACJENT] Zlecam OTWARCIE (Kolejka: 400 >= 400)
[MONITOR] Otwieram bramke nr 2
[PACJENT] Zlecam ZAMKNIECIE (Kolejka: 265 < 266)
[MONITOR] Zamykam bramke nr 2
[PACJENT] Zlecam OTWARCIE (Kolejka: 400 >= 400)
[MONITOR] Otwieram bramke nr 2
[PACJENT] Zlecam ZAMKNIECIE (Kolejka: 265 < 266)
[MONITOR] Zamykam bramke nr 2
[PACJENT] Zlecam OTWARCIE (Kolejka: 400 >= 400)
[MONITOR] Otwieram bramke nr 2
[PACJENT] Zlecam ZAMKNIECIE (Kolejka: 265 < 266)
[MONITOR] Zamykam bramke nr 2
[PACJENT] Zlecam OTWARCIE (Kolejka: 400 >= 400)
[MONITOR] Otwieram bramke nr 2
[PACJENT] Zlecam ZAMKNIECIE (Kolejka: 265 < 266)
[MONITOR] Zamykam bramke nr 2
[PACJENT] Zlecam OTWARCIE (Kolejka: 400 >= 400)
[MONITOR] Otwieram bramke nr 2
[PACJENT] Zlecam ZAMKNIECIE (Kolejka: 265 < 266)
[MONITOR] Zamykam bramke nr 2
[PACJENT] Zlecam OTWARCIE (Kolejka: 400 >= 400)
```

System utrzymuje drugie okienko otwarte mimo spadku poniżej progu otwarcia 400, zamykając je dopiero po osiągnięciu dolnego progu 266.

T6: Procedura nagłej ewakuacji (SIGINT)

Cel: Weryfikacja poprawności mechanizmu "Snapshota" (zamrożenia stanu pamięci) oraz zgodności liczby pacjentów przy nagłym przerwaniu symulacji (Ctrl+C).

Wyniki:



- Stan w momencie przerwania: Generator zablokował pamięć i wykonał zrzut stanu: 800 pacjentów wewnątrz SOR oraz 118 w kolejce (łącznie 918 procesów).
- Proces usuwania: Wysłano sygnał SIGTERM. Suma kodów wyjścia procesów (waitpid) wyniosła 800, co idealnie pokrywa się z liczbą pacjentów zajmujących zasoby (800 wew.). Pozostałe 118 procesów (kolejka) zwróciło 0.
- Raport: Sekcja "RAPORT EWAKUACJI" wyświetliła poprawne dane (800/118), zgodne ze stanem faktycznym.
- Wniosek: Mechanizm działa prawidłowo. Wyeliminowano ryzyko wyścigu (race condition), a każdy proces został poprawnie zidentyfikowany i rozliczony.
- Dygresja: teoretycznie moglibyśmy zwracać wartość inną niż zero dla osob przed poczekalnią jednak musielibyśmy zadbać o to, żeby generowanie wieku pacjenta i zapis do pamięci dzielonej o pobycie przed poczekalnia odbywały się jak najszybciej. Przyznanie wieku mogłoby się odbywać z poziomu fork() i exec() w generatorze - przekazywalibyśmy wiek jako argument (przy odpowiedniej konwersji na stringa) oraz przypisywali atoi(argv[1]) do wieku. Jednak co z pacjentami, którzy nie uaktualnili StanSOR - przed_poczekalnia++. Użycie semctl z GETNCNT również nie rozwiąże problemu, gdyż ten traktuje rodzica z dzieckiem jako pojedynczy proces. To samo się tyczy logiki semctl i GETVAL na semaforze generatora w połączeniu z GETVAL semafora poczekalni - znowu różnica zwróci nam jedynie liczbę procesów (bez rozróżnienia na dorosły / dziecko z opiekunem)