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 obecna_kolejka = stan->dlugosc_kolejki_rejestracji;
int obecny_rozkaz = stan->wymuszenie_otwarcia;

if (obecna_kolejka >= PROG_OTWARCIA && obecny_rozkaz == 0)
{
    stan->wymuszenie_otwarcia = 1;
    zapisz_raport(RAPORT_1, semid, "[PACJENT] Zlecam OTWARCIE (Kolejka: %d >= %d)\n",
                  obecna_kolejka, PROG_OTWARCIA);
}
else if (obecna_kolejka < PROG_ZAMKNIECIA && obecny_rozkaz == 1)
{
    stan->wymuszenie_otwarcia = 0;
    zapisz_raport(RAPORT_1, semid, "[PACJENT] Zlecam ZAMKNIECIE (Kolejka: %d < %d)\n",
                  obecna_kolejka, PROG_ZAMKNIECIA);
}
```

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`) jest prosty – w pętli odbiera komunikaty od pacjentów i odsyła je z powrotem (z ustawionym `mtype` na PID pacjenta):

główna pętla rejestracji: rejestracja.c

```
// rejestracja.c - główna pętla
while(!koniec_pracy)
{
    ssize_t status = msgrcv(msgid_we, &pacjent, sizeof(pacjent) - sizeof(long),
                            -2, IPC_NOWAIT);

    if (status == -1) {
        if (errno == ENOMSG || errno == EINTR) {
            usleep(50000);
            continue;
        }
        break;
    }

    pacjent.mtype = pacjent.pacjent_pid;
    if(msgsnd(msgid_we, &pacjent, sizeof(pacjent) - sizeof(long), 0) != -1)
    {
        if(!koniec_pracy) {
            zapisz_raport(KONSOLA, semid, "[Rejestracja %d] Pacjent %d -> POZ\n",
                          nr_okienka, pacjent.pacjent_pid);
        }
    }
}
```

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 zgodnie ze schematem:

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

Na każdym etapie `mtype` jest odpowiednio modyfikowany:

```
// pacjent.c - wysyłanie do rejestracji
msg.mtype = vip ? TYP_VIP : TYP_ZWYKLY;  // 1 lub 2
// ...
msgsnd(rej_msgid, &msg, sizeof(msg)-sizeof(long), 0);
msgrcv(rej_msgid, &msg, sizeof(msg)-sizeof(long), mpid, 0);  // Czeka na swój PID

// Po rejestracji - wysyłanie do POZ
msg.mtype = 1;  // POZ nie rozróżnia priorytetów
msgsnd(poz_id, &msg, sizeof(msg)-sizeof(long), 0);
msgrcv(poz_id, &msg, sizeof(msg)-sizeof(long), mpid, 0);

// Po triażu - wysyłanie do specjalisty
if (msg.typ_lekarza > 0) {
    msg.mtype = msg.kolor;  // 1=czerwony (najwyższy), 2=żółty, 3=zielony
    msgsnd(qid, &msg, sizeof(msg)-sizeof(long), 0);
    msgrcv(qid, &msg, sizeof(msg)-sizeof(long), mpid, 0);
}
```

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);
}

// Użycie przy komunikacji z rejestracją:
lock_limit(SLIMIT_REJESTRACJA);
msgsnd(rej_msgid, &msg, sizeof(msg)-sizeof(long), 0);
msgrcv(rej_msgid, &msg, sizeof(msg)-sizeof(long), mpid, 0);
unlock_limit(SLIMIT_REJESTRACJA);
```

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", "lekarz", "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:

pętla główna lekarz POZ: lekarz.c

```
// lekarz.c - funkcja praca_poz()
void praca_poz(int msgid_poz)
{
    KomunikatPacjenta pacjent;
    while(!koniec_pracy)
    {
        if(msgrcv(msgid_poz, &pacjent, sizeof(pacjent) - sizeof(long), 0, IPC_NOWAIT) == -1) {
            if (errno == ENOMSG || errno == EINTR) { usleep(50000); continue; }
            break;
        }

        // Losowanie koloru zgodnie z rozkładem statystycznym
        int r = rand() % 100;
        if (r < 10) pacjent.kolor = CZERWONY;       // 10%
        else if (r < 45) pacjent.kolor = ZOLTY;     // 35%
        else if (r < 95) pacjent.kolor = ZIELONY;   // 50%
        else {
            // 5% - odesłanie do domu
            pacjent.typ_lekarza = 0;
            pacjent.skierowanie = 1;
            pacjent.kolor = 0;
        }

        // Przypisanie specjalisty
        if (pacjent.kolor != 0) {
            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;
        msgsnd(msgid_poz, &pacjent, sizeof(pacjent) - sizeof(long), 0);
    }
}
```

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:

pętla główna lekarza specjalisty: lekarz.c

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

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

        // Odbiór pacjenta z priorytetem
        if(msgrcv(msgid, &pacjent, sizeof(pacjent) - sizeof(long), -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;
        msgsnd(msgid, &pacjent, sizeof(pacjent) - sizeof(long), 0);
    }
    shmdt(stan);
}
```

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 (3 sekundy z możliwością przerwania)
    for(int i=0; i<30; i++) {
        if(koniec_pracy) break;
        usleep(100000);  // 100ms
    }

    // 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;
}
```

Pobyt na oddziale jest realizowany jako seria krótkich `usleep()` zamiast jednego długiego `sleep()`, co pozwala na szybką reakcję na sygnał zakończenia pracy.

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) {
        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) {
    // Zakończenie wątku opiekuna
    if (potrzebny_rodzic && rodzic_utworzony) {
        pthread_cancel(rodzic_thread);
        pthread_join(rodzic_thread, NULL);
    }

    // Zwrot kodu wyjścia dla weryfikacji
    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. Wątki w procesie pacjenta (symulacja opiekuna)

I.1. Tworzenie wątku opiekuna dla nieletnich

watek rodzic: pacjent.c

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

if (wiek < 18) {
    potrzebny_rodzic = 1;
    sem_op_miejsca = 2;  // Zajmuje 2 miejsca w poczekalni
    pthread_create(&rodzic_thread, NULL, watek_rodzic, NULL);
    rodzic_utworzony = 1;
    zapisz_raport(KONSOLA, semid, "[Pacjent %d] Utworzono (Wiek: %d, VIP: %d, Opiekun TID: %lu)\n",
                  mpid, wiek, vip, (unsigned long)rodzic_thread);
}
```

I.2. Funkcja wątku opiekuna

funkcja watku rodzica

```
// pacjent.c - wątek opiekuna
void* watek_rodzic(void* arg) {
    while(1) {
        sleep(1);
        pthread_testcancel();  // Punkt anulowania
    }
    return NULL;
}
```

Wątek jest pasywny - jego jedynym zadaniem jest reprezentowanie obecności opiekuna. `pthread_testcancel()` umożliwia bezpieczne przerwanie wątku przez `pthread_cancel()`.

I.3. Zajmowanie podwójnego miejsca w poczekalni

```
// pacjent.c - wejście do poczekalni
struct sembuf wejscie = {SEM_MIEJSCA_SOR, -sem_op_miejsca, SEM_UNDO};
while (semop(semid, &wejscie, 1) == -1) {
    if (errno == EINTR) continue;
    exit(1);
}
```

O tym już była mowa przy okazji _exit() przez pacjenta podczas ewakuacji.

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

I.4. Kończenie wątku opiekuna

```
// pacjent.c - zakończenie normalne
if (potrzebny_rodzic && rodzic_utworzony) {
    pthread_cancel(rodzic_thread);
    pthread_join(rodzic_thread, NULL);
}

struct sembuf wyjscie = {SEM_MIEJSCA_SOR, sem_op_miejsca, SEM_UNDO};
semop(semid, &wyjscie, 1);
```

`pthread_cancel()` wysyła żądanie anulowania, a `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

Stała	Wartość / maksymalna przepustowość	Opis
PROG_OTWARCIA	MAX_PACJENTOW/2	Próg otwarcia drugiego okienka
PROG_ZAMKNIECIA	MAX_PACJENTOW/3	Próg zamknięcia drugiego okienka

---

Testy

Test 1: Dynamiczna Bramka nr 2

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

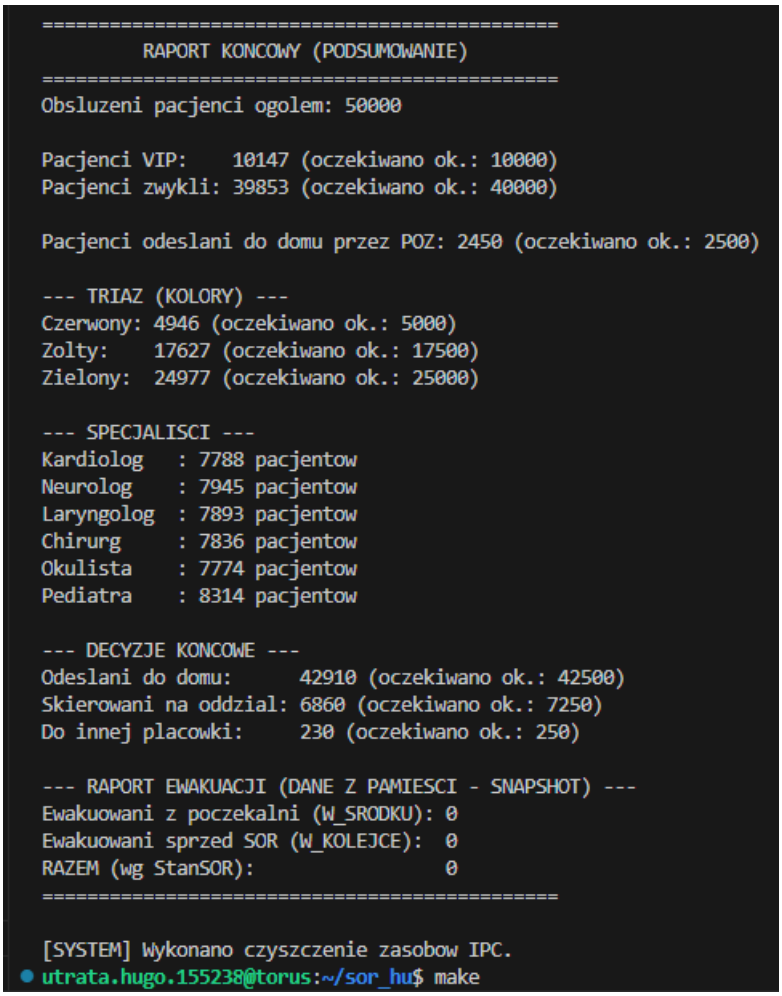
Analiza pliku `monitor_bramek.txt` pokazuje poprawne działanie mechanizmu monitorującego otwarcia bramek:

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.

Test 2: Weryfikacja statystyczna i spójność danych



Cel testu: Weryfikacja poprawności synchronizacji procesów oraz spójności danych statystycznych przy dużym obciążeniu systemu (50 000 pacjentów).

- Test sprawdza, czy suma obsłużonych pacjentów zgadza się z liczbą wygenerowanych procesów (czy żaden proces nie został "zgubiony") oraz czy rozkład losowy (triaż, decyzje) mieści się w założonych granicach prawdopodobieństwa.
- **Wyniki testu:** Wygenerowano pacjentów: 50 000
- Obsłużono ogółem: 50 000
- Suma decyzji końcowych (Odesłani do domu + Skierowani na oddział + Do innej placówki):$42910 + 6860 + 230 = 50 000
- Wniosek: Bilans pacjentów jest idealny. Żaden proces nie został utracony ani pominięty w statystykach.
- Pacjenci odesłani przez POZ do domu: 2 450
- Pacjenci skierowani do specjalistów: 50 000 - 2 450 = 47 550
- Suma pacjentów przyjętych przez specjalistów (Kardiolog...Pediatra): 7788 + 7945 + 7893 + 7836 + 7774 + 8314 = 47 550
- **Wniosek:** Przekazywanie pacjentów na linii POZ -> Specjalista działa poprawnie.
- **Rozkład statystyczny:** Wyniki rzeczywiste są bardzo zbliżone do wartości oczekiwanych (np. Czerwony: 4946 vs oczekiwano ok. 5000). Niewielkie odchylenia są naturalnym efektem działania generatora liczb losowych (rand()) i mieszczą się w normie.

Test 3: Wezwanie na oddział wszystkich specjalistów

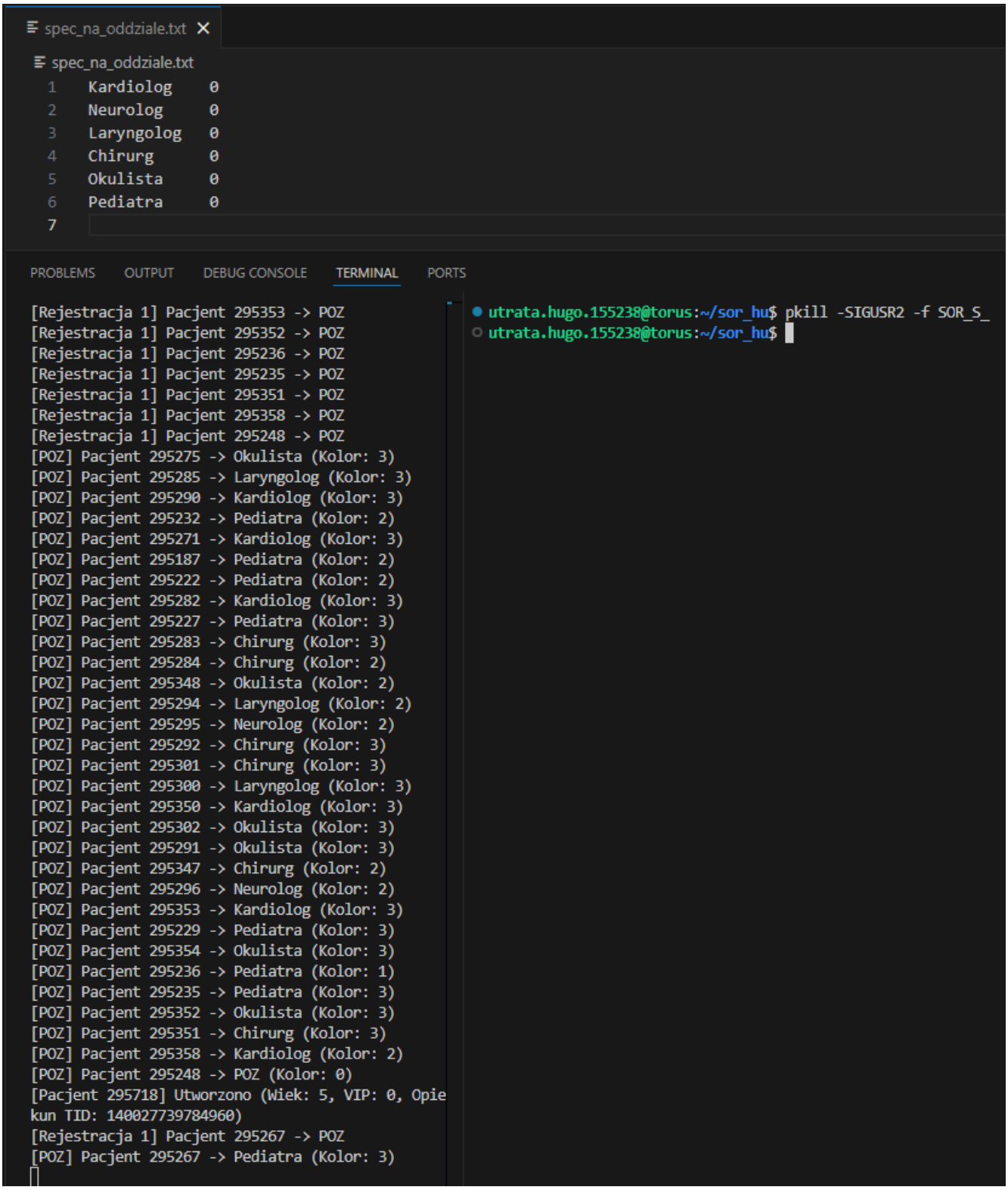
Cel testu: Weryfikacja odporności systemu na sygnał SIGUSR2 (symulacja wezwania lekarza na oddział przez Dyrektora). Test ma na celu sprawdzenie sytuacji ekstremalnej, w której wszyscy specjaliści jednocześnie przerywają pracę na SOR, udają się na oddział (symulowane przez sleep), a następnie wracają do obsługi kolejki.

Metodyka: W trakcie pełnego obciążenia systemu, w osobnej konsoli wykonano polecenie systemowe wysyłające sygnał do wszystkich procesów specjalistów jednocześnie:

*pkill -SIGUSR2 -f SOR_S_*

Flaga -f SOR_S_ pozwala na uchwycenie wszystkich procesów potomnych, których nazwa zaczyna się od zdefiniowanego w kodzie prefiksu (np. SOR_S_Kardiolog, SOR_S_Chirurg itd.).

Wyniki testu:



Reakcja na sygnał:

- Procesy specjalistów poprawnie odebrały sygnał (przerwanie funkcji systemowej msgrcv z błędem EINTR).
- Mechanizm obsługi błędów w pętli głównej lekarz.c zadziałał poprawnie – procesy nie zakończyły działania, lecz przeszły do procedury obsługi wezwania.

Zachowanie systemu (Pauza):

- Zaobserwowano chwilowe wstrzymanie logów od specjalistów (symulacja 10-sekundowego obchodu na oddziale).
- W tym czasie procesy Rejestracja oraz POZ działały nadal, kolejkując pacjentów do momentu zapełnienia kolejek IPC.

Powrót do normalnej pracy:

- Po upływie czasu symulacji oddziału, wszyscy specjaliści wznowili pobieranie komunikatów z kolejek.
- System "nadrobił" zaległości, przetwarzając nagromadzonych pacjentów.
- Symulacja zakończyła się w sposób naturalny, osiągając limit pacjentów, a raport końcowy (zgodny z Testem 2) potwierdził, że żaden pacjent nie został zgubiony w trakcie tego manewru.

Test 4: Procedura nagłej ewakuacji (SIGINT)

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

Wyniki:

```
=== ROZPOCZYNAM EWAKUACJE SOR ===
[Pacjent 346676] Utworzono (Wiek: 47, VIP: 0)
[GENERATOR] Snapshot: W srodku=800, Przed SOR=118
[GENERATOR] Zabijam pacjentow (SIGTERM)...

[GENERATOR] Ewakuacja zakonczona.
[GENERATOR] Suma kodow wyjscia (kontrolna): 800

==========================================
        RAPORT KONCOWY (PODSUMOWANIE)
==========================================
Obsluzeni pacjenci ogolem: 11815

Pacjenci VIP:    2416 (oczekiwano ok.: 2363)
Pacjenci zwykli: 9399 (oczekiwano ok.: 9452)

Pacjenci odeslani do domu przez POZ: 591 (oczekiwano ok.: 591)

--- TRIAZ (KOLORY) ---
Czerwony: 1163 (oczekiwano ok.: 1182)
Zolty:    4063 (oczekiwano ok.: 4135)
Zielony:  5998 (oczekiwano ok.: 5908)

--- SPECJALISCI ---
Kardiolog   : 1801 pacjentow
Neurolog    : 1854 pacjentow
Laryngolog  : 1802 pacjentow
Chirurg     : 1830 pacjentow
Okulista    : 1891 pacjentow
Pediatra    : 2046 pacjentow

--- DECYZJE KONCOWE ---
Odeslani do domu:      10113 (oczekiwano ok.: 10043)
Skierowani na oddzial: 1649 (oczekiwano ok.: 1713)
Do innej placowki:     53 (oczekiwano ok.: 59)

--- RAPORT EWAKUACJI (DANE Z PAMIESCI - SNAPSHOT) ---
Ewakuowani z poczekalni (W_SRODKU): 800
Ewakuowani sprzed SOR (W_KOLEJCE):  118
RAZEM (wg StanSOR):                 918
==========================================

[SYSTEM] Wykonano czyszczenie zasobow IPC.
utrata.hugo.155238@torus:~/sor_hu$
```

- 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)

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
```