Scenariusz obrony – nowy algorytm szeregowania w MINIX 2.0.3

Wstęp teoretyczny i cel

Wprowadzono trzy poziomy priorytetu dla procesów użytkownika (grupy 0, 1, 2), co wymusza kolejność wykonywania: grupa 0 (RR) > grupa 1 (aging) > grupa 2 (SJF). Dla każdej grupy dobrano osobny kwant czasu, a procesy mogą zgłaszać swój priorytet oraz deklarowany czas wykonania przez nowe wywołania systemowe SETGROUP i SETBURST.

Analiza zmian (krok-po-kroku)

Architektura – co dodano globalnie (w prostych słowach)

- **Dodatkowe pola procesu (`proc.h`):** zapamiętujemy grupę (`group`), licznik starzenia (`age`) i czasy SJF (`sjf_burst` , `sjf_rem`). Planista ma te dane „pod ręką".
- **Różne kwanty dla grup (`system.c`):** tablica `quants[3] = {5, 3, 1}` – grupa 0 dostaje najdłuższy kwant, grupa 2 najkrótszy.
- **Nowe wywołania systemowe:** `SETGROUP` i `SETBURST` (numery 78/79). MM przekazuje je do SYSTASK, który zmienia pola procesu w jądrze.

Przykład kodu (`proc.h`):

```
int group;           /* 0=RR, 1=Aging, 2=SJF */
unsigned long age;   /* licznik dla aging */
long sjf_burst;      /* zadeklarowany czas SJF */
long sjf_rem;        /* pozostały czas SJF */
```

Przykład kodu (`system.c` – kwanty):

```
unsigned short quants[3] = {5, 3, 1}; /* mnożone przez SCHED_RATE */
```

`include/minix/callnr.h` (nagłówki wywołań systemowych)

- **Lokalizacja:** `/usr/include/minix/callnr.h` (w repozytorium: `minix_usr-2/include/minix/callnr.h`).
- **Co zostało zmienione:** Podniesiono `NCALLS` do 80 i zdefiniowano numery wywołań `SETGROUP` (78) i `SETBURST` (79).
- **Jak to działa:** Nowe numery są widoczne dla przestrzeni użytkownika i mapowane w tablicach menedżera pamięci.
- **Dlaczego tak:** Dodanie nowych usług MM wymaga rezerwacji numerów wywołań systemowych, aby biblioteka `_syscall` mogła przygotować poprawne wiadomości.

`include/minix/com.h` (kody SYSTASK)

- **Lokalizacja:** `/usr/include/minix/com.h` (`minix_usr-2/include/minix/com.h`).
- **Co zostało zmienione:** Dodano kody komunikatów `SYS_SETGROUP` i `SYS_SETBURST` (22, 23) obsługiwane przez SYSTASK oraz zadeklarowano tablicę `quants[3]` dla kwantów czasowych grup.
- **Jak to działa:** MM wysyła `_taskcall` do SYSTASK z tymi kodami; system task dispatchuje je do nowych handlerów w jądrze.
- **Dlaczego tak:** Warstwa SYSTASK jest miejscem, w którym jądro wykonuje operacje administracyjne na deskryptorach procesów.

`src/mm/table.c` , `src/mm/proto.h` , `src/mm/misc.c` , `src/mm/main.c`

- **Lokalizacja:** `/usr/src/mm/…` (`minix_usr-2/src/mm/...`).
- **Co zostało zmienione:**
  - `table.c` : W tablicy `call_vec` dodano obsługę `do_setgroup` i `do_setburst` .
  - `proto.h` : Dopisano prototypy nowych funkcji.
  - `misc.c` : Zaimplementowano wrappery `do_setgroup` / `do_setburst` , które budują wiadomość i wołają `_taskcall(SYSTASK, SYS_SETGROUP/SETBURST, …)` .
  - `main.c` : Dodano `stdio.h` (logowanie/pomocnicze I/O).
- **Jak to działa:** Wywołanie z poziomu użytkownika trafia do MM (numer syscallu 78/79), a MM przekazuje żądanie do SYSTASK. Dla SJF przenoszony jest zadeklarowany „burst".
- **Dlaczego tak:** MM pośredniczy w walidacji podstawowych parametrów i jest zgodny z architekturą MINIX (serwer MM zarządza procesami użytkownika).

`src/kernel/system.c`

- **Lokalizacja:** `/usr/src/kernel/system.c` (`minix_usr-2/src/kernel/system.c`).
- **Co zostało zmienione:**
  - Dodano tablicę `quants[3] = {5, 3, 1}` (mnożone przez `SCHED_RATE`).
  - Funkcje `do_setgroup` i `do_setburst` obsługujące kody `SYS_SETGROUP/SYS_SETBURST` . Wyszukiwanie procesu po PID (`findProcDescriptor`), walidacja grupy, zapis `group` , `sjf_burst` , `sjf_rem` .
  - `do_fork` inicjalizuje pole `group` dziecka na 0.
- **Jak to działa:** SYSTASK po otrzymaniu komunikatu aktualizuje deskryptor procesu; dla SJF przechowywany jest przewidywany czas (`sjf_rem`) wykorzystywany przy wyborze procesu.
- **Dlaczego tak:** Tylko kod jądra może modyfikować kolejkę gotowych procesów i atrybuty wpływające na planistę.

Fragment `do_setgroup` (proste ustawienie grupy):

```
if (m_ptr->m1_i2 < 0 || m_ptr->m1_i2 > 2) return EFAULT;
rp = findProcDescriptor(m_ptr->m1_i1);
if (rp == NULL) return EFAULT;
rp->group = m_ptr->m1_i2;
return OK;
```

Fragment `do_setburst` (deklaracja czasu SJF):

```
rp = findProcDescriptor(pid);
if (rp == NULL) return ESRCH;
rp->sjf_burst = b;
rp->sjf_rem   = b;
return OK;
```

`src/kernel/proc.h`

- **Lokalizacja:** `/usr/src/kernel/proc.h` (`minix_usr-2/src/kernel/proc.h`).
- **Co zostało zmienione:** Struktura `proc` otrzymała pola `group`, `age`, `sjf_burst`, `sjf_rem`; zadeklarowano `extern unsigned short quants[3];`.
- **Jak to działa:**
  - `group` identyfikuje klasę (0=RR, 1=Aging, 2=SJF).
  - `age` służy do wyboru najstarszego procesu w grupie 1.
  - `sjf_*` przechowują zadeklarowany czas dla SJF.
- **Dlaczego tak:** Planista potrzebuje trwałych metadanych per-proces, aby realizować różne polityki w jednej kolejce użytkownika.

`src/kernel/main.c`

- **Lokalizacja:** `/usr/src/kernel/main.c` (`minix_usr-2/src/kernel/main.c`).
- **Co zostało zmienione:** Inicjalizacja wszystkich deskryptorów procesów ustawiająca `group = 0`.
- **Jak to działa:** Domyślnie każdy nowy proces startuje w grupie RR, dopóki nie poprosi o inną klasę.
- **Dlaczego tak:** Bez jawnego ustawienia wszystkie procesy mogłyby mieć niezainicjalizowany priorytet.

`src/kernel/clock.c`

- **Lokalizacja:** `/usr/src/kernel/clock.c` (`minix_usr-2/src/kernel/clock.c`).
- **Co zostało zmienione:** Odświeżanie kwantu `sched_ticks` zależy od `quants[group]` procesu na początku kolejki użytkownika.
- **Jak to działa:** Po wyczerpaniu kwantu zegar nadaje nową wartość proporcjonalną do klasy priorytetu (grupa 0 dostaje najdłuższy czas).
- **Dlaczego tak:** Różne polityki potrzebują różnych długości kwantu, aby zachować hierarchię (np. SJF krótszy kwant).

Fragment:

```
if (--sched_ticks == 0) {
    if (rdy_head[USER_Q] != NIL_PROC)
        sched_ticks = SCHED_RATE * quants[rdy_head[USER_Q]->group];
    else
        sched_ticks = SCHED_RATE;
    prev_ptr = bill_ptr;
}
```

`src/kernel/proc.c`

- **Lokalizacja:** `/usr/src/kernel/proc.c` (`minix_usr-2/src/kernel/proc.c`).
- **Co zostało zmienione:**
  - Nowe pomocnicze funkcje: `user_insert_tail`, `user_pick_next`, `user_move_to_head`.
  - `ready()` ustawia `age=0` i wstawia proces do kolejki użytkownika, zachowując kolejność grup (0 przed 1, 1 przed 2).
  - `user_pick_next()` wybiera: (a) jeśli na czele stoi grupa 0 – ona biega RR; (b) w grupie 1 wybierany jest proces o największym `age`; (c) w grupie 2 proces o najmniejszym `sjf_rem`. Wybrany element przenoszony jest na początek kolejki.
  - `sched()` po wyczerpaniu kwantu przerzuca bieżący proces na koniec swojej grupy i ponownie wywołuje `pick_proc()`.
- **Jak to działa:** Jedna kolejka `USER_Q` jest podzielona logicznie na trzy segmenty; wybór następnego procesu respektuje kolejność grup oraz wewnętrzną politykę (RR/aging/SJF).
- **Dlaczego tak:** Pozwala to zachować prostotę jednej kolejki użytkownika przy jednoczesnym obsłużeniu trzech różnych strategii szeregowania.

Szczegóły kolejkowania użytkownika (krok po kroku)

1. **Wstawianie (`user_insert_tail`)**
   - Gdy kolejka jest pusta – proces staje się head i tail.
   - Jeśli w kolejce są procesy tej samej grupy, nowy jest dołączany tuż za ostatnim z tej grupy (utrzymanie ciągłych segmentów).
   - Jeśli grupa=0: trafia na sam początek kolejki.
   - Jeśli grupa=1: jest wstawiany za segmentem grupy 0, ale przed segmentem grupy 2.
   - Grupa=2: domyślnie na koniec całej kolejki.
2. **Wybór (`user_pick_next`)**

- Jeśli head ma grupę 0: zwraca head (RR wewnątrz segmentu utrzymuje się dzięki `sched()`, które rotuje).
- W grupie 1: przegląd całej kolejki, wybór procesu z największym `age`, jego przesunięcie na head (reset `age` do 0).
- W grupie 2: przegląd kolejki, wybór procesu z najmniejszym `sjf_rem`, przesunięcie na head.

3. **Rotacja po kwancie (`sched`)**
   - Sprawdza, czy bieżący proces to head `USER_Q`; jeśli tak, zdejmuje go z head, przestawia na koniec swojego segmentu (przez ponowne `user_insert_tail`) i ponownie wybiera proces.
   - Dla grupy 1 resetuje `age` procesu, który oddał kwant.
4. **Konsekwencja:** Kolejność grup jest zachowana (0 > 1 > 2), a wewnątrz grup zaimplementowano odpowiednio RR, starzenie przez największe `age` i SJF przez najmniejsze `sjf_rem`.

Przykład kodu (szukanie najstarszego w grupie 1):

```
best = NIL_PROC; best_prev = NIL_PROC; best_age = 0;
prev = NIL_PROC; p = rdy_head[USER_Q];
while (p != NIL_PROC) {
    if (p->group == 1 && (best == NIL_PROC || p->age > best_age)) {
        best = p; best_prev = prev; best_age = p->age;
    }
    prev = p; p = p->p_nextready;
}
```

> Uwaga: `age` jest zerowane przy wejściu do kolejki i po użyciu kwantu. Brak mechanizmu zwiększania `age` w trakcie oczekiwania powoduje, że w obecnym kodzie grupa 1 zachowuje się jak FIFO (pierwszy proces tej grupy w kolejce będzie wybierany). Prawdziwe starzenie wymagałoby inkrementacji `age` – najprościej w obsłudze zegara w `clock.c` (np. podczas dekrementacji `sched_ticks` przejrzeć `USER_Q` i zwiększać `age` procesów z grupy 1).

`src/kernel/clock.c` i `src/kernel/proc.c` – długość kwantu i rotacja

- **Interakcja:** Długość kwantu (`sched_ticks`) jest mnożona przez `quants[group]`, a po jego zużyciu `sched()` przestawia proces na odpowiednie miejsce, co łącznie daje hierarchię priorytetów.

`src/kernel/dmp.c`

- **Lokalizacja:** `/usr/src/kernel/dmp.c` (`minix_usr-2/src/kernel/dmp.c`).
- **Co zostało zmienione:** W wydruku tablicy procesów dodano kolumnę z numerem grupy.
- **Jak to działa:** Pole `group` jest drukowane przed PID, co ułatwia diagnostykę nowego planisty.
- **Dlaczego tak:** Wizualna weryfikacja przynależności procesów do grup w trakcie testów.

`tmp/proces.c`

- **Lokalizacja:** `minix_usr-2/tmp/proces.c` (program testowy w przestrzeni użytkownika).
- **Co zostało zmienione/dodane:** Narzędzie demonstrujące nowe syscall'e. Udostępnia tryby `rr`, `aging`, `sjf`, `mix`, `onesjf` oraz wrappery `setgroup` / `setburst` korzystające z numerów 78/79.
- **Jak to działa (ścieżka danych):**
  1. W `setgroup` / `setburst` budowana jest wiadomość m1_i1=PID, m1_i2=group/burst; `_syscall(MM, SETGROUP/SETBURST, &m)` kieruje ją do MM.
  2. MM woła `_taskcall(SYSTASK, SYS_SETGROUP/SYS_SETBURST, …)`.
  3. SYSTASK aktualizuje pola `group` / `sjf_*` procesu.
  4. Dziecko w `child_worker` ewentualnie ustawia grupę/burst, drukuje log, a następnie wykonuje `burn_ticks(runtime)` (aktywne zużywanie CPU do zadanej liczby tików) lub pętlę nieskończoną.
  5. `delay_ticks` używa `times()` do odczekania między startami procesów aging, aby zasymulować starzenie.
- **Dlaczego tak:** Umożliwia ręczne uruchamianie scenariuszy testowych i obserwację kolejności przełączeń.

Jak uruchomić testy i co znaczą argumenty

- `rr N [sekundy]` – uruchamia N procesów w grupie 0 (round-robin). Domyślnie 5 sekund runtime, można podać własny czas w sekundach jako trzeci argument.
- `aging N [sekundy] [opóźnienie_ms]` – uruchamia N procesów w grupie 1. Każdy ma runtime w sekundach (domyślnie 5), między startami wprowadzane jest opóźnienie w milisekundach (domyślnie 300 ms), co pozwala zobaczyć starzenie.
- `sjf [s=sekundy] B1 B2 ...` – uruchamia procesy grupy 2 (SJF). `s=` ustawia runtime w sekundach (domyślnie 5). Kolejne argumenty to zadeklarowane bursty (np. 30 90 150). Kolejność wykonania powinna zaczynać się od najmniejszego burstu.
- `mix [sekundy]` – uruchamia mieszankę: jeden RR, jeden aging oraz trzy SJF z burstami 30/90/150. Parametr w sekundach ustawia runtime (domyślnie 5).
- `onesjf` – ustawia bieżący proces do grupy 2 i deklaruje burst=30; szybki test konfiguracji.

Przykłady (do uruchomienia w MINIX po skompilowaniu `proces`):

- `./proces rr 3 5`
- `./proces aging 3 5 300`
- `./proces sjf s=5 30 90 150`
- `./proces mix 5`

Przepływ sterowania (flow)

1. **Użytkownik** wywołuje `setgroup` / `setburst` (np. z `proces.c`), co przez `_syscall` generuje przerwanie do jądra i wiadomość do **MM** z numerem 78/79.
2. **MM** (tablica `call_vec`) deleguje do `do_setgroup` / `do_setburst`, które budują komunikat m1 i wysyłają go `_taskcall` do **SYSTASK** z kodem

`SYS_SETGROUP` lub `SYS_SETBURST`.

3. **SYSTASK** (`system.c`) odbiera wiadomość, wyszukuje deskryptor procesu po PID, waliduje parametry, zapisuje `group`, `sjf_burst`, `sjf_rem` i korzysta z globalnej tablicy `quants`.
4. Przy **przygotowaniu do biegu** `ready()` wstawia proces do kolejki użytkownika zgodnie z jego grupą, zeruje `age` i zachowuje kolejność segmentów grup.
5. **Zegar (`clock.c`)** po każdym kwancie ustawia `sched_ticks = SCHED_RATE * quants[group]` dla pierwszego procesu w `USER_Q`.
6. **Planista (`pick_proc()` + `user_pick_next()`)** wybiera: najpierw zadania i serwery, potem użytkowników z kolejką trójsegmentową (RR dla grupy 0, aging w grupie 1, SJF w grupie 2).
7. **Kontext powrotny:** Po wyczerpaniu kwantu `sched()` przenosi bieżący proces w ramach jego segmentu i wywołuje `pick_proc()`; reszta systemu pozostaje niezmieniona.

Scenariusze testowe (min. 4)

Przykładowe wywołania programu `proces` (po skompilowaniu w systemie MINIX):

1. **RR – równy podział:** `./proces rr 3 5`
   Trzy procesy w grupie 0, każdy powinien krążyć round-robin z długim kwantem (quants[0]).
2. **Aging – nadrabianie czekania:** `./proces aging 3 5 300`
   Trzy procesy w grupie 1 z opóźnionym startem; najstarszy w kolejce powinien być wybierany w pierwszej kolejności.
3. **SJF – najkrótsze zadanie:** `./proces sjf s=5 30 90 150`
   Cztery procesy w grupie 2 o różnych zadeklarowanych burstach; kolejność powinna zaczynać się od burst=30.
4. **Mieszany priorytet:** `./proces mix 5`
   Jednoczesne uruchomienie grup 0, 1 oraz kilku z grupy 2 – grupa 0 zawsze przed 1, a 1 przed 2; wewnątrz grupy 2 krótsze bursty wcześniej.
5. (Opcjonalnie) **Pojedyncze SJF:** `./proces onesjf` – szybka weryfikacja ustawienia grupy/bursta dla bieżącego PID.

Podsumowanie implementacji

Kluczowe elementy to dwie nowe usługi jądra (SETGROUP/SETBURST), rozszerzona struktura `proc` z metadanymi planisty, dynamiczne kwanty zależne od grupy oraz pojedyncza kolejka użytkownika podzielona logicznie na trzy segmenty z różnymi politykami wyboru. Najważniejszym wyzwaniem było spięcie przepływu wiadomości (użytkownik → MM → SYSTASK → `proc`), tak aby planista mógł korzystać z przekazanych parametrów przy zachowaniu zgodności z architekturą MINIX.