

Gra w życie

Dokumentacja projektu

Programowanie systemowe

Autorzy

Krzysztof Czerenko

Paweł Froń

1. Opis zagadnienia	3
2. Opis działania	4
3. Sposób działania	5
4. Wykorzystane konstrukcje programowania systemowego	6
4.1 Wątki	6
4.1.1 Tworzenie wątków	6
4.1.2 Kończenie pracy wątków	6
4.2 Zmienne warunkowe	7
4.2.1 Inicjalizacja	7
4.2.2 Czekanie na warunek	7
4.2.3 Wybudzanie wątków	7
4.2.4 Niszczanie zmiennej warunkowej	8
4.3 Mutexy	8
4.3.1 Inicjalizacja mutexa	8
4.3.2 Blokowanie mutexa	9
4.3.3 Zwalnianie mutexa	9
4.3.4 Niszczanie mutexa	9
4.4 Dostęp do plików	9
4.4.1 Otwieranie pliku	9
4.4.2 Czytanie z pliku	10
4.4.3 Pisanie do pliku	10
4.4.4 Zamykanie pliku	10
5. Sposób użycia konstrukcji w projekcie	10
5.1 Wykorzystanie wątków	11
5.2 Wykorzystanie zmiennych warunkowych i mutexów	11
5.3 Wykorzystanie obsługi plików	12

1. Opis zagadnienia

Gra w życie ("Game of Life") to automat komórkowy wymyślony przez brytyjskiego matematyka Johna Hortona Conwaya. Po raz pierwszy opisano jej zasady w artykule z 1970 roku. Zasady działania tego automatu są następujące:

- gra toczy się na dwuwymiarowej, nieskończonej, kwadratowej siatce
- każda komórka może znajdować się w dwóch stanach ("żywy" lub "martwy")
- każda komórka oddziałuje ze swoimi ośmioma sąsiadami (według sąsiedztwa Moore'a)
- w każdym kroku czasowym ("generacji"), wszystkie komórki są aktualizowane zgodnie z zasadą:
 - żywa komórka umiera, jeśli ma mniej niż dwóch, lub więcej niż trzech sąsiadów
 - martwa komórka ożywa, jeśli ma dokładnie trzech sąsiadów
 - w przeciwnym wypadku, komórka pozostaje w swoim poprzednim stanie

Na cele implementacji, oczywiście nie mogliśmy zastosować nieskończonej siatki, dlatego jej rozmiary są podawane jako parametry do programu. Przyjeliśmy, że wszystkie komórki znajdujące się poza tak ustalonymi granicami, są martwe.

2. Opis działania

Aby uruchomić program należy skompilować plik .c, a następnie użyć komendy:

```
./<plik programu> <plik wejściowy> <plik wyjściowy>  
<liczba wierszy> <liczba kolumn> <liczba generacji>
```

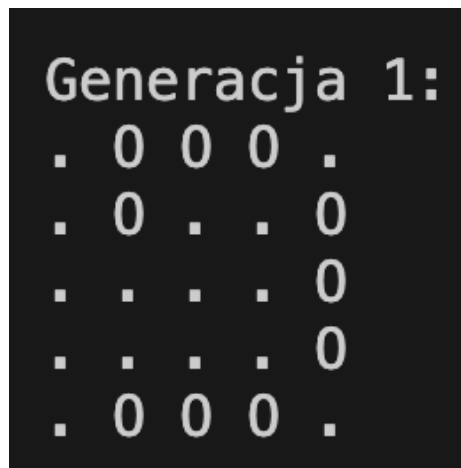
Gdzie:

plik wejściowy - plik ze startową planszą

plik wyjściowy - plik do którego zapisywany jest stan planszy po kolejnych iteracjach

Program wizualizuje kolejne generacje na ekranie konsoli, a także zapisuje je w pliku wyjściowym.

Poniżej znajduje się przykład wizualizacji planszy:



```
Generacja 1:  
. 0 0 0 .  
. 0 . . 0  
. . . . 0  
. . . . 0  
. 0 0 0 .
```

3. Sposób działania

Nasz program umożliwia symulowanie dowolnej wielkości planszy przez dowolną z góry określoną liczbę generacji.

Stan początkowy planszy można wczytać z przygotowanego wcześniej pliku tekstowego, w którym 0 oznacza komórkę martwą, a 1 komórkę żywą.

Aby przyspieszyć obliczenia program dzieli obliczenia na wiele wątków i każdy wątek dostaje określoną liczbę wierszy do przetwarzania. Liczba wątków jest określona funkcją $\text{MIN}(\text{liczba wierszy}, \text{liczba dostępnych procesorów logicznych})$. Gwarantuje to maksymalną wydajność, gdyż wszystkie wątki pracują niezależnie.

Do synchronizacji wątków stosujemy zmienne warunkowe `pthread_cond_t` z biblioteki `<pthread.h>`. Ich działanie opiszemy dokładnie niżej.

Dodatkowo stosujemy mechanizmu mutexów do zabezpieczania dostępu do współdzielonych pomiędzy procesami zmiennych takich jak numer generacji czy licznik wątków, które skończyły pracę.

Wynikiem działania programu jest wyświetlana w terminalu symulacja, a także zapis stanów po kolejnych generacjach w pliku wyjściowym.

4. Wykorzystane konstrukcje programowania systemowego

4.1 Wątki

Wątki to narzędzie pozwalające na równoległe wykonywanie kodu. W C ich obsługa jest zaimplementowana w bibliotece POSIX Threads (pthread).

Do podstawowych funkcji związanych z obsługą wątków należą:

4.1.1 Tworzenie wątków

```
int pthread_create(pthread_t *thread, const  
pthread_attr_t *attr, void *(*start_routine)(void *),  
void *arg)
```

Funkcja ta tworzy nowy wątek gdzie:

*pthread_t *thread* – wskaźnik na identyfikator wątku

*pthread_attr_t *attr* – atrybuty wątku (domyślnie null)

*void *(*start_routine)(void *)* – wskaźnik na funkcję, którą wątek ma wykonywać

*void *arg* – argumenty funkcji

4.1.2 Kończenie pracy wątków

```
int pthread_join(pthread_t thread, void **retval);
```

Funkcja ta blokuje wykonanie programu, aż podany wątek nie zakończy działania:

pthread_t thread – identyfikator wątku

*void **retval* – wskaźnik do przechwycenia wartości zwracanej przez wątek (jeżeli brak to null)

4.2 Zmienne warunkowe

Do synchronizacji wątków z wątkiem głównym w kolejnych generacjach używamy mechanizmu zmiennych warunkowych z tej samej biblioteki.

Zmienne `pthread_cond_t` pozwalają na uśpienie wątku do momentu, kiedy nie otrzyma on sygnału wybudzenia. Używa się ich, kiedy jakiś wątek musi poczekać na zakończenie jakiejś czynności przez inny wątek.

Wykorzystuje się je często wraz mutexami.

4.2.1 Inicjalizacja

Taką zmienną można zainicjować:

- dynamicznie:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER
```

- za pomocą funkcji:

```
pthread_cond_init(&cond, NULL);
```

4.2.2 Czekanie na warunek

```
pthread_cond_wait(pthread_cond_t *cond,  
pthread_mutex_t *mutex)
```

Funkcja ta usypia wątek do momentu otrzymania sygnału wybudzenia i zwalnia dany mutex. Po wybudzeniu wątek ponownie spróbuje on ponownie zablokować mutex.

4.2.3 Wybudzanie wątków

```
pthread_cond_signal(pthread_cond_t *cond)
```

Wybudza pojedynczy wątek czekający na ten sygnał.

```
pthread_cond_broadcast(pthread_cond_t *cond)
```

Wybudza wszystkie wątki czekające na ten sygnał.

4.2.4 Niszczanie zmiennej warunkowej

Służy do tego ta funkcja:

```
pthread_cond_destroy(&cond)
```

4.3 Mutexy

Mutex (mutual exclusion - wzajemne wykluczenie) to kolejny mechanizm synchronizacji będący częścią POSIX Threads.

Zapewnia, że do danego zasobu, np. zmiennej w danym czasie dostęp ma wyłącznie jeden wątek.

Zapobiega to tak zwanemu wyścigowi o zasoby (race condition).

Zasada działania jest prosta

- wątek, który chce uzyskać dostęp do zasobu musi zablokować mutex
- a po skorzystaniu odblokować

4.3.1 Inicjalizacja mutexa

Można to zrobić:

- statycznie:

```
pthread_mutex_init(&mutex, NULL)
```

-dynamicznie:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
```


4.3.2 Blokowanie mutexa

```
pthread_mutex_lock(&mutex)
```

Wątek po wywołaniu tej funkcji:

- jeżeli mutex jest wolny, przejmie blokadę i będzie mógł skorzystać z zasobu
- jeżeli nie to wątek czeka na zwolnienie blokady

4.3.3 Zwalnianie mutexa

```
pthread_mutex_unlock(&mutex);
```

Wywołując tą funkcję wątek zwalnia mutex i co za tym idzie zasób.

4.3.4 Niszczanie mutexa

```
pthread_mutex_destroy(&mutex);
```

Ta funkcja niszczy mutex.

4.4 Dostęp do plików

Do obsługi biblioteki używamy funkcji podstawowej biblioteki C stdio.h.

4.4.1 Otwieranie pliku

Do otworzenia pliku używa się:

```
FILE *fopen(const char *filename, const char *mode)
```

Przyjmuje ona:

- `const char *filename` - nazwę pliku do otwarcia
- `const char *mode` - tryb w którym chcemy otworzyć plik

Zwraca natomiast wskaźnik do pliku.

4.4.2 Czytanie z pliku

Używaliśmy do tego:

```
int fgetc(FILE *stream)
```

Funkcja ta przyjmuje wskaźnik do pliku, a zwraca znak char jako int.

4.4.3 Pisanie do pliku

Do tego wykorzystaliśmy:

```
int fputc(int character, FILE *stream)
```

Funkcja ta przyjmuje znak, który chcemy zapisać i wskaźnik do pliku i zwraca zapisany znak.

4.4.4 Zamykanie pliku

Do tego służy funkcja:

```
int fclose(FILE *stream)
```

5. Sposób użycia konstrukcji w projekcie

Tu opiszemy, gdzie wykorzystaliśmy opisane wyżej konstrukcje.

5.1 Wykorzystanie wątków

W naszym projekcie tworzone są wątki, którym przydzielany jest zakres wierszy, które mają przetwarzać w kolejnych generacjach. Ich identyfikatory przechowywane są w tablicy. Każdy wątek wykonuje funkcję `worker_thread`.

5.2 Wykorzystanie zmiennych warunkowych i mutexów

Mamy dwie zmienne globalne sterujące symulacją, do których dostęp jest zarządzany przez mutexy.

Jest to:

`current_gen` - obecnie przetwarzana generacja

`done_count` - licznik wątków, które zakończyły przetwarzania aktualnej generacji

Wątek główny działa w pętli for po generacjach:

```
for (int gen = 1; gen <= generations; gen++)
```

Na początku każdej generacji blokujemy mutex i ustawiamy

`current_gen` :

```
pthread_mutex_lock(&cond_mutex);  
current_gen = gen;
```

Następnie startujemy wszystkie wątki, które są uśpione:

```
pthread_cond_broadcast(&cond_start);
```

I czekamy dopóki wszystkie wątki nie skończą pracy:

```
while (done_count < num_workers) {  
    pthread_cond_wait(&cond_done,  
    &cond_mutex);  
}
```

Następnie resetujemy licznik `done_count` i odblokowujemy mutex:

```
done_count = 0;
pthread_mutex_unlock(&cond_mutex);
```

W wątkach przetwarzających planszę natomiast:

Blokujemy mutex i czekamy na kolejną generację:

```
pthread_mutex_lock(&cond_mutex);
while (current_gen == local_gen) {
    pthread_cond_wait(&cond_start, &cond_mutex);
}
```

Sprawdzamy czy ta generacja nie jest ostatnią:

```
if (current_gen > generations) {
    pthread_mutex_unlock(&cond_mutex);
    break;
}
```

Aktualizujemy lokalny numer generacji i odblokowujemy mutex:

```
local_gen = current_gen;
pthread_mutex_unlock(&cond_mutex);
```

5.3 Wykorzystanie obsługi plików

Z pliku wczytujemy początkowy stan planszy, a także zapisujemy do pliku planszę po kolejnych generacjach.