Automaty

Programowanie współbieżne MIM UW 2017/18

Filip Binkiewicz 332069

Opis rozwiązania podzielę na trzy sekcję, każdą dotyczącą jednego z trzech programów `tester`, `validator` oraz `run`. W każdej z tych sekcji podam zastosowany model współbieżności, sposób komunikacji, oraz poczynione założenia dotyczące bezpieczeństwa programu.

- ### Run
    - Model współbieżności

    Struktura procesów w programie `run` odpowiada strukturze przebiegu automatu na danym słowie. Dla każdego zbioru przejść typu `(q, a) -> (q1, .., qk)` funkcja `fork` wywoływana jest `k` razy. Procesy odpowiadające stanowi bez przejść (lub stanowi osiągniętemu po przeczytaniu całego słowa) nie tworzą podprocesów.
    Innymi słowy, każdy stan w przebiegu automatu reprezentowany jest przez pojedynczy proces, przy czym proces główny odpowiada stanowi początkowemu. Każdy proces świadomy jest opisu automatu, przez co może zinterpretować wyniki przekazane od swoich potomków.
    - Komunikacja

    Do przekazywania wyników pomiędzy spokrewnionymi procesami programu `run` wykorzystane są łącza nienazwane (tzw. `pipe`). Do komunikacji z tworzącymi proces `run` procesem `validator` wykorzystane są dwie kolejki komunikatów (`message queues`) - jedna do przeczytania opisu automatu, druga do wysłania odpowiedzi. Kolejki te nazwane są `"{vpid}_in"` oraz `"{vpid}_out"`, gdzie `vpid` to identyfikator procesu walidatora, który stworzył podproces `run`.
    - Poczynione założenia

    Nie zakładamy powodzenia wywołania systemowego `fork`. W razie błędu przesyłany jest sygnał do procesu głównego. Niestety nie byłem w stanie zapewnić bezpieczeństwa przed tzw. fork bombą. Gdy automat jest za duży, taki model może doprowadzić do przepełnienia pamięci (wtedy program powinien zawiesić się). Przykład takiego automatu znajduje się w pliku `examples/maxtomat.txt`

- ### Tester
    - Model współbieżności

    Program tester wykonuje się w jednym procesie.
    - Komunikacja

    Program `tester` komunikuje się z programem `validator` za pomocą nazwanych kolejek komunikatów. Na początku, aby proces `validator` stworzył podproces dedykowany dla danego testera, wysyła wiadomość do głównej kolejki, na której

nasłuchuje proces główny programu `validator`. Następnie program `tester` przesyła słowa na kolejkę o nazwie `"{tpid}_in"` oraz oczekuje na odpowiedź na kolejce `"{tpid}_out"`, gdzie `tpid` to identyfikator procesu testera.

- Poczynione założenia

W przypadku błędu w którymś wywołaniu systemowym (np. `mq_open`, `mq_send`, `mq_receive`) deskryptory wszystkich kolejek są zamykane, a pliki specjalne kolejek `{tpid}_in` oraz `{tpid}_out` są usuwane z dysku. Aby zapewnić poprawne funkcjonowanie programu, zakładamy, że liczba testerów nie przekracza 10 (wartość ta może być większa, w przypadku przekroczenia limitu zgłaszany jest błąd `mq_open: too many files open`.

- Validator

- Model współbieżności

Proces główny programu `walidator` oczekuje w pętli na wiadomości przesyłane kolejką nazwaną. Walidator akceptuje dwa rodzaje wiadomości:

1. Prośba o utworzenie podprocesu dla testera
2. Zakończenie programu (!)

W przypadku otrzymania prośby, program główny tworzy nowy proces, który będzie komunikował się poprzez dwie kolejki z programem tester (ich opis znaleźć można w sekcji `tester`). Tak stworzony podproces oczekuje na słowa od swojego testera, a po otrzymaniu słowa przekazuje je nowo stworzonemu procesowi `run`.

- Komunikacja

Proces główny programu `validator` utrzymuje dwie globalne kolejki:

1. Kolejkę do obsługi nowych testerów
2. Kolejkę do odebrania podsumowań od swoich podprocesów.

Dodatkowo każdy podproces utrzymuje dwie kolejki dla testera oraz po dwie kolejki dla każdego stworzonego programu `run`.

- Poczynione założenia

Nie zakładamy nic o wynikach wywołań funkcji systemowych. W razie błędu dowolny podproces programu `validator` wysyła specjalny sygnał obsługiwany przez proces główny. Następnie proces główny wysyła sygnały do podprocesów, które z kolei wysyłają ten sam sygnał do testerów.