Systemy operacyjne

Lista zadań nr 10

Na zajęcia 15 i 16 grudnia 2021

Należy przygotować się do zajęć czytając następujące materiały: [2, 11.1 – 11.5, 12.8 – 12.10], [5, 26, 27 i 33].

UWAGA! W trakcie prezentacji należy być gotowym do zdefiniowania pojęć oznaczonych **wytłuszczoną** czcionką.

Zadanie 1. Czym różni się **przetwarzanie równoległe** (ang. *parallel*) od **przetwarzania współbieżnego** (ang. *concurrent*)? Czym charakteryzują się **procedury wielobieżne** (ang. *reentrant*)? Podaj przykład procedury w języku C (a) wielobieżnej, ale nie **wątkowo-bezpiecznej** (ang. *thread-safe*) (b) na odwrót. Kiedy w jednowątkowym procesie uniksowym może wystąpić współbieżność?

Wskazówka: Właściwości *thread-safe* i *async-signal-safe* są opisane również w POSIX Safety Concepts[1].

Zadanie 2. Wybierz odpowiedni scenariusz zachowania wątków, w którym konkurują o dostęp do zasobów, i na tej podstawie precyzyjnie opisz zjawisko **zakleszczenia** (ang. *deadlock*), **uwięzienia** (ang. *livelock*) oraz **głodzenia** (ang. *starvation*). Dalej rozważmy wyłącznie ruch uliczny. Kiedy na jednym lub wielu skrzyżowaniach może powstać każde z wymienionych zjawisk? Zaproponuj metodę (a) **wykrywania i usuwania** zakleszczeń (b) **zapobiegania** zakleszczeniom. Rozważ dwa warianty: w pierwszym kierowcy patrzą na kierującego ruchem policjanta, w drugim kierowcy mogą przekazywać sobie znaki rękoma. Czy policjant mógłby zagłodzić pewnych kierowców?

Zadanie 3. W poniższym programie występuje **sytuacja wyścigu** (ang. *race condition*) dotycząca dostępów do współdzielonej zmiennej «tally». Wyznacz jej najmniejszą i największą możliwą wartość.

```
1 const int n = 50;
2 shared int tally = 0;
3
4 void total() {
5   for (int count = 1; count <= n; count++)
6     tally = tally + 1;
7 }
8
9 void main() { parbegin (total(), total()); }
```

Dyrektywa «parbegin» rozpoczyna współbieżne wykonanie procesów. Maszyna wykonuje instrukcje arytmetyczne wyłącznie na rejestrach – tj. kompilator musi załadować wartość zmiennej «tally» do rejestru, przed wykonaniem dodawania. Jak zmieni się przedział możliwych wartości zmiennej «tally», gdy wystartujemy k procesów zamiast dwóch? Odpowiedź uzasadnij pokazując przeplot, który prowadzi do określonego wyniku.

Zadanie 4. Podaj odpowiedniki funkcji `fork(2)`, `exit(3)`, `waitpid(2)`, `atexit(3)` oraz `abort(3)` dla wątków i opisz ich semantykę. Porównaj zachowanie wątków **złączalnych** (ang. *joinable*) i **odczepionych** (ang. *detached*). Zauważ, że w systemie Linux procedura `pthread_create` odpowiada za utworzenie reprezentacji wątku w przestrzeni użytkownika, w tym utworzenie stosu i uruchomienie wątku przy pomocy wywołania `clone(2)`. Kto zatem odpowiada za usunięcie segmentu stosu z przestrzeni użytkownika, gdy wątek zakończy pracę? Pomocne może być zajrzenie do implementacji funkcji `pthread_exit` i `pthread_join`.

[1] https://www.gnu.org/software/libc/manual/html_node/POSIX-Safety-Concepts.html

Zadanie 5. Implementacja wątków POSIX skomplikowała semantykę niektórych zachowań procesów, które omawialiśmy do tej pory. Co nieoczekiwanego może się wydarzyć w wielowątkowym procesie uniksowym gdy:

- jeden z wątków zawoła funkcję `fork(2)` lub `execve(2)` lub `exit_group(2)`?
- proces zadeklarował procedurę obsługi sygnału «SIGINT», sterownik terminala wysyła do procesu «SIGINT» – w kontekście którego wątek zostanie obsłużony sygnał?
- określono domyślną dyspozycję sygnału «SIGPIPE», a jeden z wątków spróbuje zapisać do rury `pipe(2)`, której drugi koniec został zamknięty?
- czytamy w wielu wątkach ze pliku zwykłego korzystając z tego samego deskryptora pliku?

Zadanie 6. Na podstawie [2, 14.4] opowiedz jaka motywacja stała za wprowadzeniem wywołania `poll(2)` [3, 63.2.2] do systemów uniksowych? Czemu lepiej konstruować oprogramowanie w oparciu o `poll`, niż o odpytywanie deskryptorów albo powiadamianie o zdarzeniach bazujące na sygnale «SIGIO»? Na jakich plikach można oczekiwać na zdarzenia przy pomocy `poll(2)` [3, 63.2.3]? Czemu wszystkie deskryptory przekazywane do `poll` powinno skonfigurować się do pracy w trybie nieblokującym? Jak zapewnić, żeby wywołania `connect(2)`, `accept(2)`, `read(2)` i `write(2)` na gnieździe sieciowym zawsze zwróciły «EWOULDBLOCK» zamiast blokować się w jądrze [7, 16]? Chcemy by po wywołaniu `poll(2)` pierwsza instancja wyżej wymienionych wywołań systemowych nie zwróciła «EWOULDBLOCK». Jaka wartość musi być wpisana przez jądro do pola «revents» struktury «pollfd» dla danego deskryptora pliku, żeby mieć tę pewność?

Ściągnij ze strony przedmiotu archiwum «so21_lista_10.tar.gz», następnie rozpakuj i zapoznaj się z dostarczonymi plikami. **UWAGA!** Można modyfikować tylko te fragmenty programów, które zostały oznaczone w komentarzu napisem «TODO».

Zadanie 7. Program «echoclient-thread» wczytuje plik tekstowy zadany z wiersza poleceń i dzieli go na linie zakończone znakiem '\n'. Następnie startuje podaną liczbę wątków, z których każdy wykonuje w pętli: nawiązanie połączenia, wysłanie «ITERMAX» losowych linii wczytanego pliku, zamknięcie połączenia. Wątki robią to tak długo, aż do programu nie przyjdzie sygnał «SIGINT».

Twoim zadaniem jest tak uzupełnić kod, by program uruchamił «nthreads» wątków i poczekał na ich zakończenie. Czemu nie można go łatwo przerobić w taki sposób, żeby główny wątek czekał na zakończenie pozostałych wątków tak, jak robi się to dla procesów przy pomocy `wait(2)`?

Zadanie 8. Program «echoserver-poll» implementuje serwer usługi «echo» przy pomocy wywołania `poll(2)`, ale bez użycia wątków i podprocesów. W kodzie wykorzystano wzorzec projektowy **pętli zdarzeń** (ang. *event loop*) do współbieżnej obsługi wielu połączeń sieciowych. Program «echoserver-select» robi to samo, ale przy pomocy wywołania `select(2)`.

Twoim zadaniem jest uzupełnienie pliku źródłowego «echoserver-poll.c». W pętli zdarzeń należy obsłużyć połączenia przychodzące, nadejście nowych danych na otwartych połączeniach i zamknięcie połączenia. Do przetestowania serwera użyj programu «echoclient-thread» z poprzedniego zadania.

Literatura

[1] „*Computer Systems: A Programmer's Perspective*"
Randal E. Bryant, David R. O'Hallaron
Pearson Education Limited; 3rd edition; 2016

[2] „*Advanced Programming in the UNIX Environment*"
W. Richard Stevens, Stephen A. Rago
Addison-Wesley Professional; 3rd edition; 2013

[3] „*The Linux Programming Interface: A Linux and UNIX System Programming Handbook*"
Michael Kerrisk
No Starch Press; 1st edition; 2010

[4] „*Systemy operacyjne*"
Andrew S. Tanenbaum, Herbert Bos
Helion; wydanie czwarte; 2015

[5] „*Operating Systems: Three Easy Pieces*"
Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
https://pages.cs.wisc.edu/~remzi/OSTEP/

[6] „*Computer Networking: A Top-Down Approach*"
James F. Kurose, Keith Ross
Pearson; 8th Edition; 2021

[7] „*Unix Network Programming, Volume 1: The Sockets Networking API*"
W Richard Stevens, Bill Fenner, Andrew M. Rudoff
Addison-Wesley Professional; 3rd edition; 2003