Łukasz Klasiński

Opis pracowni z Systemów Operacyjnych

Prowadzący pracownię: Zdzisław Płoski

Wrocław, 31 stycznia 2018

1. Wstęp

Tematem pracowni jest wykorzystanie semaforów do jakiegoś problemu programistycznego. Mój program bazuje na problemie "Ucztujących filozofów". Tworzy on n procesów, z czego każdy chce otrzymać dostęp do dwóch plików, czyli widelców w oryginalnym zadaniu. Siedzą oni przy "stole", więc dwa sąsiednie procesy dzielą jeden plik ze sobą. Zaimplementowane jest także zasymulowane przetrzymywanie dostępu, czyli "jedzenie" oraz przerwa między nim. Dzięki temu powstaje problem, który najlepiej jest rozwiązać za pomocą semaforów.

2. Funkcje i struktury

Program można podzielić na funkcje, zmienną globalną oraz strukturę semafora.

```
1        typedef struct{
2            int val;
3            char file[20];
4            bool stolen;
5            bool dirty;
6        } semaphore;
```

Przedstawia strukturę semafora. Przeznaczenie poszczególnych pól zostanie wyjaśnione później.

```
1                static semaphore** mox;
```

Jest zmienna globalną, dzieloną między procesami, która umożliwia im dostęp do semafora.

```
int wait_s(semaphore* s, semaphore* holder)
```

Odpowiada za przydzielanie dostępu do semafora.

```
void signal_s(semaphore* s)
```

Sygnalizuje zwolnienie semafora.

```
int rand_num(int a, int b)
```

Losuje losowe liczby z podanego przedziału. Potrzebne do losowania ile dany proces ma czekać.

```
void run_process(int i, semaphore* s1, semaphore* s2,
int min, int max, int iterate)
```

Jest kodem wykonywanym przez procesy. W argumentach otrzymuje dwa semafory, które umożliwiają dostęp do pliku, minimalny i maksymalny czas czekania, które używane są przy funkcji *randnum*, oraz ile ma wykonać dostępów do pliku.

```
int main(int argc, char** argv)
```

Funkcja główna programu, która odpowiada za inicjalizację zmiennych, uruchomienie procesów oraz obsługę interfejsu.

3. Analiza kodu

3.1 Funkcja Main

```
int procn = 0;
int min = 0;
int max = 0;
int iterate = -1;
if(argc >= 4){
    procn = atoi(argv[1]);
    min = atoi(argv[2]);
    max = atoi(argv[3]);
}
else{
    printf("need at least 3 arguments");
    return EXIT_FAILURE;
}
if(argc >= 5){
    iterate = atoi(argv[4]);
}


int *forkid = (int*)malloc(sizeof(int)*procn);
```

```
20      mox = mmap(NULL, sizeof(semaphore*)*procn, PROT_READ |
    PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
```

Na początku w funkcji main inicjalizujemy zmienne początkowe, czyli:

1. procn - zmienna ta przetrzymuje ilość nowych procesów, które zostaną utworzone.
2. min - odpowiada za minimalny czas czekania procesu.
3. max - odpowiada za maksymlny czas czekania procesu.
4. iterate - mówi ile dany proces ma wykonać dostępów do plików.

Następnie wykonywane jest sprawdzenie, czy użytkownik wprowadził argumenty oraz sparsowanie ich z przedstawionymi wcześniej zmiennymi. W lini 19 inicjalizujemy zmienną *forkid*, która będzie przetrzymywać id utworzonych procesów. Będzie to potrzebne, aby je potem zamknąć. W ostatnim wierszu rezerwujemy pamięć na zmienna globalną

```
1       static semaphore** mox;
```

która jest tak naprawdę dwuwymiarową tablicą, zawierającą n struktur semafora.

```
1   for(int i = 0; i < procn; i++){
2       *(mox+i) = (semaphore*)malloc(sizeof(semaphore));
3       (*(mox+i))->val = 1;
4       sprintf((*(mox+i))->file, "file_%d.txt", i);
5       FILE *fp = fopen((*(mox+i))->file, "w+");
6       fclose(fp);
7       (*(mox+i))->stolen = false;
8       (*(mox+i))->dirty  = true;
9   }
```

Kolejny fragment kodu odpowiada za inicjalizację struktury semafora oraz utworzenie plików, do których mają umożliwiać dostęp.

```
1       for(int i = 0; i < procn; i++){
2           int pid = fork();
3           if(pid < 0){
4               exit(EXIT_FAILURE);
5           }
6           if(!pid){
7               // process
8               srand(time(NULL)+i);
9               run_process(i, *(mox+i), *(mox+(((i-1)%procn)+
    procn)%procn), min, max, iterate);
10          }
11          else{
```

```
12                  //parent
13                  forkid[i] = pid;
14              }
15          }
```

Następna pętla inicjalizuje procesy za pomocą funkcji *fork*. Sprawdza, czy otrzymane id jest nieujemne, co sygnalizuje błąd przy tworzeniu procesu, różny od zera, co znacza, że utworzony został nowy proces oraz czy jest równe zeru - wtedy wiemy, że jesteśmy w głównym procesie programu. Jeśli nastąpił błąd, program kończy proces, sygnalizując błąd. W przeciwnym przypadku inicjalizuje "seed", który będzie służyć do losowania czasu czekania przez proces, oraz uruchamiamy funkcję wykonywalną procesu. Przekazujemy jego numer, semafory oraz resztę zmiennych omawianych wcześniej. W kodzie wykonywanym przez główny proces zapamiętujemy id procesów.

```
1           char c;
2           while(true){
3               scanf("%c", &c);
4               if(c == 'e')
5               {
6                   for(int i = 0; i < procn; ++i){
7                       free(*(mox+i));
8                       kill(forkid[i], SIGKILL);
9                   }
10                  munmap(mox, sizeof(semaphore*)*procn);
11                  printf("processes killed, exiting!\n");
12                  return EXIT_SUCCESS;
13              }
14          }
```

W kolejnej sekcji oczekujemy na sygnał od uzytkownika, na zakończenie programu. Jeśli go otrzymamy, to zwalniamy pamięć oraz zabijamy utworzone procesy.

3.2 Procesy

Opiszemy funkcję

```
1           void run_process(int i, semaphore* s1, semaphore* s2,
    int min, int max, int iterate){
2               while(iterate -- != 0){
3                   usleep(rand_num(min, max));
4                   wait_s(s1, NULL);
5                   // got access to file 1
6                   while(true){
7                       if(wait_s(s2, s1))
8                           break;
```

```
                    else{
                        s1->dirty = false;
                        signal_s(s1);
                        wait_s(s1, NULL);
                    }
                }
                // got access to file 2
                usleep(rand_num(min, max));      // holding files
                FILE *fp = fopen(s1->file, "a+");
                fprintf(fp, "process %d eated this file!\n", i);
                fclose(fp);
                fp = fopen(s2->file, "a+");
                fprintf(fp, "process %d eated this file!\n", i);
                fclose(fp);
                printf("process %d eated!\n", i);
                s1->dirty = true;
                s2->dirty = true;
                if(s1->stolen)
                    s1->dirty = false;
                if(s2->stolen)
                    s2->dirty = false;
                signal_s(s1);
                signal_s(s2);
            }
            exit(EXIT_SUCCESS);
        }
```

Potrzebna jest także implementacja *wait* oraz *signal*

```
            int wait_s(semaphore* s, semaphore* holder){
                s->stolen = true;
                if(holder != NULL){
                    while(s->val <= 0){
                        if(holder->stolen && !holder->dirty)
                            return false;
                    }
                }
                else{
                    while(s->val <= 0);
                }
                s->val--;
                s->stolen = false;
                return true;
            }

            void signal_s(semaphore* s){
                s->val++;
            }
```

Proces na początku wchodzi do pętli while, która wykonuje się określoną

liczbę razy, zależnie od zmiennej *iterate*. Następnie usypiamy go na losowy czas. Kiedy się wybudzi próbuje otrzymać dostęp do pierwszego semafora. Zauważmy, że przekazujemy jeden argumen jako $NULL$, co powoduje, że funkcja *wait* wykona prostą pętlę, która czeka, aż semafor się zwolni. Po otrzymaniu dostępu następuje próba dostępu do drugiego semafora. Tym razem jest inaczej, ponieważ proces posiada już jeden. W celu wykluczenia zakleszczeń został zaimplementowany pomysł, który polega na oznaczaniu semaforów jako *dirty*, co pozwala procesom "ukraść" posiadany już semafor, jeśli jest "brudny". W taki przypadku proces musi oddać ten "widelec" i oznaczyć go jako "czysty". Zatem jak widać w kodzie wchodzimy do nieskończonej pętli, którą opuścimy dopiero po otrzymaniu dostępu do obu plików. We fragmencie

```
        while(true){
            if(wait_s(s2, s1))
            break;
        else{
            s1->dirty = false;
            signal_s(s1);
            wait_s(s1, NULL);
        }
```

,czekamy na drugi semafor, podając w drugim argumencie posiadany. Wtedy podczas czekania na dostęp w funkcji *wait*

```
            while(s->val <= 0){
                if(holder->stolen && !holder->dirty)
                    return false;
            }
```

sprawdzamy także, czy podczas oczekiwania inny proces nie oznaczył naszego semafora jako *stolen*. Wtedy jeśli semafor jest "brudny", to musimy go oddać, więc funkcja *wait* przerywa działanie i zwraca $false$, który sygnalizuje procesowi, że musi oddać posiadany semafor i spróbować jeszcze raz dostać do niego dostęp. Natomiast jeśli uda nam się zawczasu otrzymać dostęp, to zwraca *true* i pętla procesu się kończy. Następnie przetrzymujemy dostęp do plików emulując "jedzenie" z oryginalnego problemu i wpisujemy do pliku informację, że dany proces go odwiedził. Na koniec zwalniamy semafory. Należy tu jeszcze dodać, że jeśli podczas używania plików jakiś inny plik sygnalizował próbę otrzymania dostępu (poprzez pole *stolen* w semaforze), to zmieniamy pole *dirty* na $false$.

W uogólnieniu, problem zakleszczenia został rozwiązany w podobny sposób, jak zaproponowali to K. Mani Chandy i J. Misra w oryginalnym problemie ucztujących filozofów.