

Kamil Matuszewski

Systemy operacyjne – problem palaczy tytoniu

Spis treści

1	Opis zadania	3
1.1	Problem palaczy tytoniu	3
1.2	Sposób rozwiązania	3
2	Użyte programy	4
2.1	Program prog.c	4
2.2	Program agent.c	5
2.3	Program smoker.c	5
3	Testowanie	6

1. Opis zadania

1.1. Problem palaczy tytoniu

Załóżmy, że trzech palaczy chciałoby zapalić papierosa. Do zrobienia papierosa potrzebują tytoniu i papieru, a dodatkowo, muszą zapalić papierosa zapalkami. Każdy palacz ma nieograniczony dostęp do jednego z tych składników. Pozostałe dwa musi in dostarczyć **agent**. Agent wystawia na stół dwa różne, losowe składniki. **Palacz** któremu do papierosa brakuje właśnie tych dwóch składników, bierze je ze stołu, skręca papierosa i go zapala. Agent czeka aż palacz skończy palić, i ponownie, wystawia dwa różne, losowe składniki. Proces ten powtarza się w nieskończoność.

1.2. Sposób rozwiązania

Do rozwiązania zadania potrzebujemy czterech **procesów**. Są to **proces agenta** oraz trzy **procesy palaczy: palacz z tytoniem, palacz z papierem oraz palacz z zapalkami**. Dodatkowo, dla ułatwienia testowania, utworzymy **proces główny**, który będzie tworzył i wywoływał pozostałe cztery procesy. Do synchronizacji procesów użyjemy **semaforów**. Będą cztery semafony.

semaphore_agent który będzie mówił, czy agent może wyłożyć składniki, czy też nie.

semaphore_tobacco który będzie mówił, czy palacz z tytoniem powinien sięgnąć do stołu po składniki.

semaphore_paper który będzie mówił, czy palacz z papierem powinien sięgnąć do stołu po składniki.

semaphore_matches który będzie mówił, czy palacz z zapalkami powinien sięgnąć do stołu po składniki.

Proces główny utworzy, i po raz pierwszy otworzy semafony, a następnie je zamknie. Za pomocą polecenia `FORK()` utworzymy cztery procesy, które wywołają osobne programy. W rzeczywistości, skoro procesy palaczy działają analogicznie, wystarczy jeden program wywołany z odpowiednim parametrem. Dla numeru 1 będzie to palacz z tytoniem, dla numeru 2 – palacz z papierem, dla pozostałych – palacz z zapalkami. Proces agenta czeka na **semaphore_agent**, i jeśli jest dostępny, losuje dwa różne składniki, wypisuje odpowiedni komunikat i zwalnia semafor odpowiedniego palacza. Palacz natomiast, czeka na odpowiedni semafor i, jeśli jest on dostępny, wypisuje komunikat o zapaleniu papierosa, a następnie zwalnia semafor agenta. Wszystkie procesy działają w nieskończonej pętli.

2. Użyte programy

2.1. Program prog.c

W prog.c zajmujemy się utworzeniem nowych procesów: Po zadeklarowaniu odpowiednich zmiennych musimy zająć się semaforami.

```
sem_unlink("/semaphore_agent");
sem_unlink("/semaphore_tobacco");
sem_unlink("/semaphore_paper");
sem_unlink("/semaphore_matches");

semaphore_agent=sem_open("/semaphore_agent", O_CREAT, (S_IRWXU | S_IRWXG | S_IRWXO), 1);
semaphore_tobacco=sem_open("/semaphore_tobacco", O_CREAT, (S_IRWXU | S_IRWXG | S_IRWXO), 0);
semaphore_paper=sem_open("/semaphore_paper", O_CREAT, (S_IRWXU | S_IRWXG | S_IRWXO), 0);
semaphore_matches=sem_open("/semaphore_matches", O_CREAT, (S_IRWXU | S_IRWXG | S_IRWXO), 0);

sem_close(semaphore_agent);
sem_close(semaphore_tobacco);
sem_close(semaphore_paper);
sem_close(semaphore_matches);
```

Na początku musimy zwolnić semafony o podanej nazwie, jeśli takie istnieją. Służy do tego funkcja SEM_UNLINK. Potem tworzymy nowe semafony. Zrobimy to za pomocą operacji SEM_OPEN. Skoro semafony nie są utworzone, musimy je utworzyć. Zgodnie z założeniami programu, semaphore_agent ustawiamy na 1, natomiast pozostałe semafony na 0. Potem zamykamy semafony: nie będą nam już potrzebne w tym programie. Przejdźmy do samego wykonania programu:

```
if(agent==0)
{
    char* args[]={ "agent", 0 };
    char* env[]={ NULL };
    execve("agent", args, env);
}
else
{
    smoker_tobacco=fork();

    if(smoker_tobacco==0)
    {
        char* args[]={ "smoker", "1", 0 };
        char* env[]={ NULL };
        execve("smoker", args, env);
    }
    else
    {

```

Kopiujemy ten proces za pomocą funkcji FORK(). Następnie, mamy dwie opcje: albo znajdujemy się w procesie macierzystym (FORK() zwraca 1), albo w procesie potomnym (FORK() zwraca 0). Dla procesu macierzystego wykonujemy program dalej - tworząc kolejne procesy palaczy. Jeśli jesteśmy w procesie potomnym, wywołujemy program *agent.c*, którym zajmiemy się za chwilę.

Analogicznie, tworzymy palaczy, wywołując jednak program *smoker.c* z odpowiednim parametrem.

Na sam koniec, jeśli wciąż znajdujemy się w procesie-matce, uruchamiamy nieskończoną pętlę, która umożliwi nam zatrzymanie wszystkich procesów, w razie takiej potrzeby.

```
else
{
    while(1);
}
```

Jeśli jesteśmy w procesie potomnym, kończymy program.

2.2. Program agent.c

Zgodnie z założeniami, agent.c odpowiada za proces agenta.

```
first_item=(1<<(rand()%3));

do
{
    second_item=(1<<(rand()%3));
}
while(first_item==second_item);
```

Omówmy najpierw zmienne `first_item` i `second_item`. Najpierw, losujemy wartość z przedziału $[0, 2]$. O taką wartość przesuwamy binarnie jedynek - otrzymujemy w ten sposób jedną z trzech wartości: 001 010 lub 100. Ułatwi to nam sprawdzanie warunków. Wystarczy bowiem zrobić odpowiednią maskę:

```
int mask=(first_item|second_item);
```

Jeśli uznamy, że 001 odpowiada za tytoń, 010 za papier a 100 za zapalki, w `mask` będą zapalone dwa bity, odpowiadające za semafor, który powinien zostać zwolniony. Tak więc jeśli mamy $110 = 6$, `semaphore_tobacco` zostanie zwolniony, jeśli $101 = 5$, `semaphore_paper`, a jeśli $011 = 3$, zwolniony zostanie `semaphore_matches`.

2.3. Program smoker.c

Procesy wszystkich trzech palaczy zdefiniowane są w programie `smoker.c`. W zmiennej `number` przechowywać będziemy numer wywołania

```
int number = atoi(argv[1]);
```

Zależnie, czy wywołamy `smoker.c` z parametrem 0, 1 czy innym, nasz program będzie robił co innego – a raczej robił to samo, zwalniając tylko inny semafor i wypisując adekwatny komunikat. Przy wypisywaniu komunikatów używam funkcji `SLEEP()`, dzięki której zwiększam czytelność wypisywanych komunikatów.

```
if(number==1) //smoker_tobacco
{
    sem_wait(semaphore_tobacco);
    sleep(1);
    printf("Palacz z tytoniem: Dostałem papier i zapalki! Wreszcie mogę zapalić!\n");
```

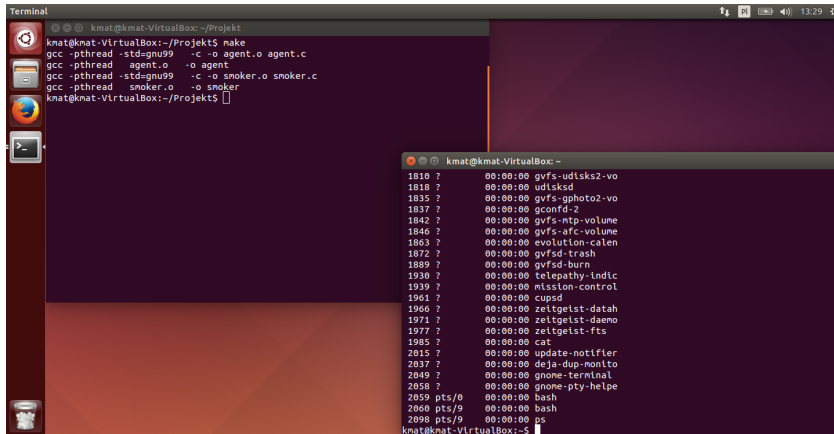
Ostatecznie zwalniamy `semaphore_agent`, umożliwiając programowi `agent.c` ponowne wyłożenie składników.

```
sem_post(semaphore_agent);
```

3. Testowanie

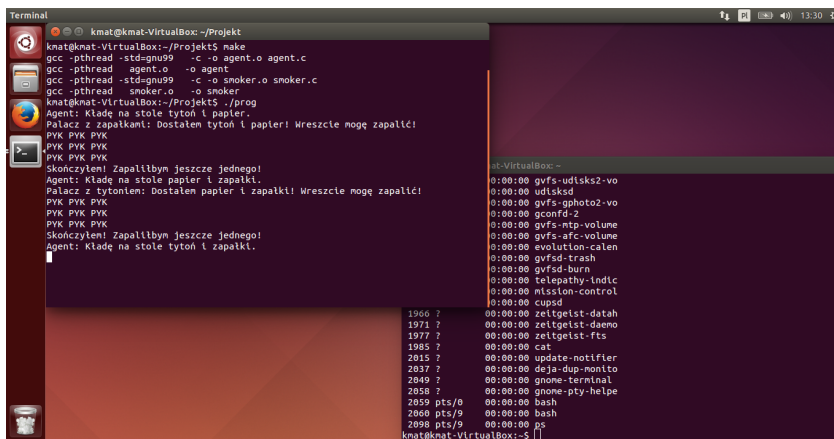
Do poprawnego uruchomienia programu wymagany jest system linux.

Przed uruchomieniem programu należy użyć polecenia **make**. Możemy też za pomocą **ps -A** sprawdzić, jakie procesy działają w tle.



```
kmat@kmat-VirtualBox:~/Projekt$ make
gcc -pthread -std=gnu99 -c -o agent.o agent.c
gcc -pthread agent.o -o agent
gcc -pthread -std=gnu99 -c -o smoker.o smoker.c
gcc -pthread smoker.o -o smoker
kmat@kmat-VirtualBox:~/Projekt$
```

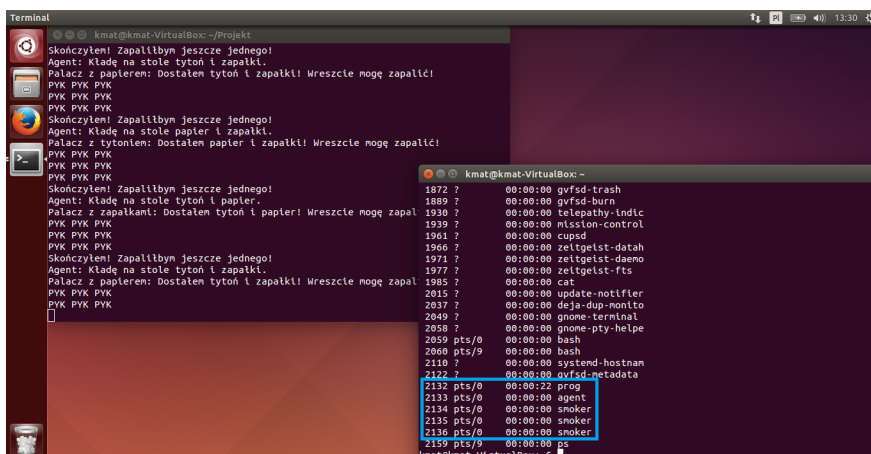
Teraz, za pomocą polecenia **./prog** możemy uruchomić nasz projekt.



```
kmat@kmat-VirtualBox:~/Projekt$ ./prog
Agent: Kładę na stole tyton i zapaliki.
Palacz z zapalnikami: Dostałem tyton i papier! Wreszcie mogę zapalić!
PYK PYK PYK
PYK PYK PYK
Skonczyłem! Zapaliłbym jeszcze jednego!
Agent: Kładę na stole papier i zapaliki.
Palacz z tytoniem: Dostałem papier i zapaliki! Wreszcie mogę zapalić!
PYK PYK PYK
PYK PYK PYK
Skonczyłem! Zapaliłbym jeszcze jednego!
Agent: Kładę na stole tyton i zapaliki.
```

Jak widzimy, program działa: wyświetlają się odpowiednie komunikaty, agent wyklada składniki a odpowiedni palacz reaguje. Nie ma także sytuacji, że agent wystawia składniki przed tym jak palacz skończy palić, bądź których palacz "blokuje" dostęp do jakiegoś składnika.

Ale czy rzeczywiście utworzyliśmy trzy procesy? Możemy to sprawdzić, uruchamiając w drugim terminalu polecenie **ps -A**



```
kmat@kmat-VirtualBox:~/Projekt$ ps -A
1810 ? 00:00:00 gvfs-udisks2-vo
1810 ? 00:00:00 udisksd
1835 ? 00:00:00 gvfs-gphoto2-vo
1837 ? 00:00:00 gconfd-2
1842 ? 00:00:00 gvfs-mtp-volume
1846 ? 00:00:00 gvfs-sftp-volume
1863 ? 00:00:00 evolution-calen
1872 ? 00:00:00 gvfsd-trash
1889 ? 00:00:00 gvfsd-burn
1930 ? 00:00:00 telepathy-indic
1939 ? 00:00:00 mission-control
1961 ? 00:00:00 cupsd
1966 ? 00:00:00 zeitgeist-datch
1971 ? 00:00:00 zeitgeist-daemo
1977 ? 00:00:00 zeitgeist-fis
1985 ? 00:00:00 cat
2015 ? 00:00:00 update-notifier
2037 ? 00:00:00 deja-dup-monito
2049 ? 00:00:00 gnome-terminal
2058 ? 00:00:00 gnome-pty-helpe
2059 pts/0 00:00:00 bash
2060 pts/9 00:00:00 bash
2098 pts/9 00:00:00 ps
kmat@kmat-VirtualBox:~$
```

Rzeczywiście, procesy zostały utworzone. Skoro tak, to program działa zgodnie z założeniami.