Temat:
System zarządzania zadaniami i ich komunikacja (Scheduler, MessageBox).

1.Uruchamianie zadań / przełączanie zadań.

Praca wielozadaniowa (multitasking), daje programiście aplikacyjnemu możliwość tworzenia oprogramowania (zadań) działającego równolegle z innymi. Teoria i sposoby realizacji wykraczają poza ramy tego dokumentu, poruszone zatem zostaną tylko najprostsze i najważniejsze zagadnienia.

Generalnie upraszczając, systemy wielozadaniowe można podzielić na:

Kooperujące - Programista musi zrobić wszystko pisząc aplikacje, aby jego zadanie było kończone najszybciej jak to jest tylko możliwe. Jeżeli jest to niemożliwe musi być przekazane sterowanie do zadania przełączającego (pozwala innym zadaniom coś wykonać), a po następnym wywołaniu podjąć przerwane zadania (np. długotrwałe, skomplikowane obliczenia, oczekiwanie na interakcje z użytkownikiem lub wolnymi urządzeniami peryferyjnymi). Z reguły takie działanie jest akceptowalne w małych i prostych systemach, wymaga od programisty dużo uwagi nad projektowaniem aplikacji.

Wywłaszczające - Programista pisze aplikacje nie wiedząc nic o tym kiedy i jak przekazywane jest sterowanie między zadaniami. Takie podejście powoduje że nie wiadomo kiedy zadanie zostanie przerwane, istnieje zatem konieczność dbania o spójność danych wykorzystywanych przez wiele procesów, wprowadzenie operacji atomowych (nie przerywalnych, wykluczających wyścig między procesami), natomiast system przełączania jest bardziej skomplikowany, gdyż konieczne jest zachowywanie kontekstu przerwanego zadania, a w odpowiednim momencie jego odtworzenie.

Proces przełączania zadań (`Scheduler`), dba o odpowiednie wywoływanie procesów zgodnie z powyższymi modelami. Kolejność wywoływania zadań może być realizowana według algorytmu "wszystkim po równo" (algorytm prosty w implementacji, lecz mało efektywny), lub z wykorzystaniem priorytetowania zadań.

2.Implementacja przełączania zadań

Podstawowym zadaniem (w ramach tego ćwiczenia) może być funkcja o prototypie:

```
void funcX(void);
```

Zakładając że zbiór zadań jest ponumerowany od `0` do `N`, mamy następujące funkcje o podobnych prototypach: `func1`, `func2`, `func3`, `...`, `funcN`. Dla takiego zestawu można utworzyć w obszarze pamięci kodu (`Code`) tablicę:

```
code void (*code f[])(void)={
  func1, func2, func3, ...
};
```

Wywołanie dowolnej tak zadeklarowanej funkcji sprowadzałoby się do wywołania:

```
(*f[ task ])();
```

Gdzie zmienna `task` jest odpowiednikiem numeru funkcji aktualnie wywoływanej. Wartość `task` nie może wskazywać (indeksować) obszaru poza danymi zapisanymi w tablicy `f.` Nieprzestrzeganie tego założenia może doprowadzić do zawieszenia systemu a w ogólności powodować błędną pracę.

Przy pomocy prostego przełącznika (`schedulera`) można po zakończeniu jednego zadania wywołać inne. Ważne jest aby funkcje były tak zapisane by trwać jak najkrócej. W szczególności nie wykonywały pętli nieskończonych lub bardzo długich (dłuższych niż czas dopuszczalny dla zadania), bez przekazywania obsługi przełącznikowi cyklicznie.

Poniższy przykład ilustruje jak unikać tworzenia pętli nieskończonych:

Oryginalny kod z pętlą nieskończoną-długą	Zmieniona wersja kodu bez pętli nieskończonej-długiej

...
...
void funcN(void){
long l; //zmienna lokalna funkcji!!!
for(l=0; l<2147483647; l++){
... //zadanie do
//wykonania wewnątrz „pętli”
}
}
...
	...
	long l=0; //zmienna globalna!!!
	void funcN(void){
	if(l<2147483647){
	... //zadanie do wykonania
	//wewnątrz „pętli”
	l++;
	}
	}
	...

Ja widać zadanie wykonywane w funkcji: funcN, będzie wykonywane tyle samo razy, różnica polegać będzie na sposobie wykonywania samej pętli. W przypadku „złym” (lewa strona tabeli) funkcja raz wywołana będzie realizować wszystkie iteracje pętli w drugim „dobrym” (prawa strona tabeli) funkcja wykona tylko jedną iterację pętli, resztę iteracji wykonane będzie przy następnych wywołaniach zadania przełącznika. W sumie obie wykonają tyle samo iteracji ale rozwiązanie dobre pozwoli innym zadaniom się wykonywać w „tle”.
Na uwagę zasługuje zmienna „l”, jest ona używana dla sterowania przebiegiem działania pętli, lecz w większości przypadków używa się struktur które przechowują stan przerwanego zadania (stan nie musi być przechowywany w jednej zmiennej), dzięki czemu następne wejście w zadanie odtwarza stan z poprzedniego wywołania.

3.Modularyzacja

Tworzenie zadań w przypadku rozbudowanych systemów może być powierzone niezależnym zespołom. W takich przypadkach, a także ze względu na przejrzystość kodu, każde z zadań dobrze jest umieszczać w innym pliku (module). Sam podział na pliki nie miałby większego sensu gdyby nie separacja obiektów. Jednym z założeń systemów wielozadaniowych jest separacja zadań – ich komunikacja jest możliwa ale realizowana jest w ściśle kontrolowany sposób (patrz pkt. 5.Komunikacja miedzy zadaniami). W uroszczeniu separacje można osiągnąć przez umieszczenie zadań (tutaj funkcji) w różnych plikach:

Plik: taska.c
 	
int ita,jta;
void TaskA(void){
...
}
	Plik: taska.h
	
	void TaskA(void);

Plik: taskb.c
 	
int itb,jtb;
void TaskB(void){
...
}
	Plik: taskb.h
	
	void TaskB(void);

Plik: taskc.c
 	
int itc,jtc;
void TaskC(void){
...
}
	Plik: taskc.h
	
	void TaskC(void);

W tak podanym zestawie plików mamy trzy zadania: TaskA, TaskB, TaskC. Pliki nagłówkowe będą użyteczne wyłącznie przełącznikowi zadań, opisany następująco:

```
#include „taska.h”
#include „taskb.h”
#include „taskc.h”

#define MAX_TASK 3

code void (*code f[])(void)={
  TaskA, TaskB, TaskC
};
```

```
void main(void){
 char task=0;
 for(;;){
   (*f[task])();
   task++;
   if(task==MAX_TASK)
     task=0;
 }
}
```

Nazwy zmiennych w plikach task...c dobrano nie przypadkowo, sugeruje iż stanowią „własność" modułu deklarującego – i nie wolno innym modułom korzystania z nich, dodatkowo nazwy pomagają wyśledzić ewentualne odstępstwa od tej reguły.

4.Przłącznik zadań z kolejka

W poprzednim punkcie zamieszczono opis trywialnego przełącznika. Działanie jego polega na ciągłym wywoływaniu zadań (tutaj funkcji) z tablicy „f". Taka praca przełącznika zakłada iż każde z zadań chce się wykonywać w nieskończoność. Rozwiązanie takie jest mało efektywne, zwłaszcza gdy zadanie nie ma więcej nic do zrobienia, lub trzeba powołać inne zadanie do życia.

Najlepszym rozwiązaniem powyższych problemów jest utworzenie kolejki zadań. Powołanie zadania do życia polegało by na włożeniu go do kolejki, np.:

```
post(TaskA);
```

Funkcja `post` jest typu:

```
char post(void (*tp) ());
```

gdzie parametr `*tp` to wskaźnik do zadania wkładanego do kolejki. Zadaniem systemu przełącznika jest ciągłe sprawdzanie stanu kolejki i uruchamianie zadania na jej szczycie, pętla główna takiego procesu mogła by wyglądać tak:

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

gdzie właściwe wywoływanie zadań miało by miejsce w funkcji: `run_task();`

Aby zapewnić sprawną kolejkę zadań wystarczy zadeklarować tablice:

```
sched_entry_T queue[MAX_TASKS];
```

gdzie elementem tej tablicy jest:

```
typedef struct {
  void (*tp) ();
} sched_entry_T;
```

I dzięki takim deklaracjom możliwe będzie włożenie do MAX_TASK zadań do wykonania (za pomocą funkcji `post`).

Dla prawidłowego działania przełącznika wymagane są jeszcze dwie zmienne:

```
unsigned char sched_full;
unsigned char sched_free;
```

gdzie `sched_full` będzie wskazywać czy jest jeszcze miejsce w kolejce, a `sched_free` wskazywać będzie na zadanie do wykonania (korzystać z niej będzie funkcja `run_task`).

Nasunąć się może pytanie a co się stanie w przypadku gdy kolejka się opróżni. Do takiej sytuacji nie powinno się dopuścić jeżeli system ma działać nieskończenie, w większości systemów zawsze działa jedno zadanie np.: `IDLE` – czyli zadanie które sprawdza czy nie istnieją warunki do rozpoczęcia innego zadania, sytuacjami takimi może być interakcja z użytkownikiem (wydanie polecenia uruchom zadanie XXX), lub zdarzenie zewnętrzne które mogło by być obsługiwane przez nowe zadanie.

Implementacja zadania `IDLE` (dalej nazywanego `TaskA`), mogła by wyglądać tak:

```
void TaskA(void){
 if(sprawdzeni_warunków_wywołania_TaskB()==0)
      post(TaskB);
 if(sprawdzeni_warunków_wywołania_TaskC()==0)
      post(TaskC);
 post(TaskA);
}
```

Przy tak napisanym zadaniu, wystarczy aby system przełącznika przed uruchomieniem pętli sprawdzającej kolejkę włożył zadanie A na jej początek, np.:

```
void main(void){
 sched_init();
 post(zadanieA);
  while(1) {
    run_task();
  }
}
```

Gdzie funkcja `sched_init()` przygotowywałaby kolejkę do pracy przełącznika. Zastanawiający może być zapis w kodzie zadania `TaskA`, wpisanie na jego końcu `post(TaskA)`, jest to niezbędne aby zadanie działało w nieskończoność. Działanie systemu przełącznika powinna tak być realizowane aby włożenie zadania `A` trafiało na koniec kolejki tak aby dać szansę zadaniom `B` i `C` się wykonać. Bardziej zaawansowane systemy realizują też priorytetowanie kolejki,

5.Komunikacja miedzy zadaniami

Idea systemów wielozadaniowych zakłada możliwość przesyłania informacji między procesami (po to by np. synchronizować zadania, dzielić się danymi). Adresowanie zadań nie jest ogólnie znormalizowane. W najprostszym podejściu zakłada się że każde zadanie ma swój numer - adres. Każde z zadań ma swoją kolejkę (w prostszych systemach kolejka może mieć długość 1), do której inne zadania (za pomocą systemu operacyjnego) będą wkładać wiadomości.
Podstawowe funkcje mogą być następujące:

`SendMSG` -Wyślij wiadomość (wstawienie do kolejki zadania adresata), argumentem musi być adres procesu i wiadomość.

`ReciveMSG` -Obierz wiadomość z kolejki (aktualnego zadania), jeżeli jest ona pusta nie czekaj, zwróć kod błędu (np.: `QUEUE_EMPTY`).

`WaitReciveMSG` -Czekaj aż w kolejce (aktualnego zadania) pojawi się wiadomość i pobierz ja, zakłada się wystąpienie sytuacji w której możliwe jest czekanie w nieskończoność (aż wiadomość się pojawi).

Powyższe funkcje są implementowane przez system operacyjny, co z tym się wiąże podczas np. wykonywania funkcji `WaitReciveMSG` system może nie przekazać sterowania do funkcji wywołującej aż do otrzymania wiadomości dla wywołującego procesu (po to by nie wywoływać zadania czekającego - powodowałoby to marnotrawienie czasu procesora).
Przykładowa struktura wiadomości może mieć strukturę:

```
struct MSG{
  t_adres sender;       //kto wysłał wiadomość
  t_msg text;           //wiadomość właściwa
};
```

Gdzie typy:

`t_adres` -typ reprezentujący adres, np.: numer procesu (dla małych systemów np. takie które są oparte o procesor `80C51` warto ograniczyć liczbę procesów do 255, wtedy można podstawić za `t_adres` typ `unsigned char`, np.:

```
#define t_adres unsigned char
```

`t_msg` -typ zmiennej trzymający informacje o adresie (wskaźnik) wiadomości lub samą wiadomość (tablica), wybór zależy od konkretnej aplikacji - mocno zależy od ilości pamięci dostępnej w systemie i od długości wiadomości, np.:

```
#define t_msg unsigned int
```

Dla uproszczenia można stosować jedną kolejkę dla wszystkich zadań, bardziej skomplikowane staną się wtedy procedury `ReciveMSG`, `WaitReciveMSG`, kosztem uproszczenia `SendMSG`.
Jedną z wiadomości może być informacja `END_PROCESS` (przekazywaną w polu `text`), po otrzymaniu której proces ma w możliwe krótkim czasie zwolnić wszelkie zasoby i wywołać funkcje `exit()` - czyli przekazać sterowanie programowi przełączającemu (`scheduler`) bez ponownego wywoływania.

6.Uwagi końcowe

W prostych systemach należy ograniczać ilość procesów – głównie z powodu problemu wielkości stosu i obszaru zmiennych.
Projektując system przełączania zadań z wywłaszczaniem, nie wolno zapomnieć o czasie przełączania. Im ten czas będzie większy (w porównaniu z czasem działania zadań), tym efektywność systemu mocno spadnie. Podobnie im czas w jakim zadanie może się wykonywać będzie za długi - inne zadania będą miały wrażenie "zawieszania" się systemu.
W przypadku systemów kooperujących sytuacja jest jeszcze bardziej skomplikowana - każde zadanie musi samo oszacować co może zrobić w ogólnie przyjętym przedziale czasu (nie istnieje arbiter przełączający niezależny od zaintersowanych). Wszystko zatem zależy od "wzajemnej uprzejmości" zadań, od wyobraźni i przezorności programisty piszącego dane zadanie.
Konstruowanie przełącznika z wywłaszczaniem wymaga wnikania w zagadnienia wewnętrznej budowy procesora po to aby efektywnie (liczy się czas przełączania) i poprawnie (zapamiętanie stanu musi dotyczyć też obiektów nie dostępnych z poziomu języka „C”).
Bardzo często takie przełączanie może dość mocno ograniczyć liczbę procesów mogących działać równocześnie w systemie. W systemach o małych zasobach liczba procesów może być ograniczana do dziesiątek a nawet jedności (np.: 4-6).
W systemach wielozadaniowych istnieje także problem dzielenia się pamięcią. Nie możliwa jest poprawna praca wielu procesów bez wyraźnego podzielenia zasobów dostępnych dla każdego z nich.
Większe systemy dysponują specjalnymi mechanizmami wspieranymi sprzętowo do kontroli uprawnień dostępu do zasobów.