[[si:lab:lab9]]

Multitasking în Sisteme Embedded

Tipuri de multitasking

Deoarece discutăm în principal sisteme embedded cu resurse restrânse, nu vom discuta despre procese, ci despre task-uri sau fire de execuție. Un fir de execuție este o secvență de cod care rulează pe procesor cu un anumit context (hint: pentru o singură instrucțiune, de ce vor depinde execuții diferite ale acestei instrucțiuni?). În cazul sistemelor uniprocesor (care încă reprezintă o majoritate în domeniul embedded), multi-tasking-ul este implementat ca o multiplexare în timp a diferitelor fire de execuție.

După felul în care firele de execuție se succed pe procesor, multi-tasking-ul poate fi de două feluri:

Colaborativ

Task-urile sunt conștiente de faptul că împart procesorul, schimbarea unui fir cu altul se face prin *cedarea* procesorului (yield). Dat fiind că task-ul este conștient că întrerupe execuția, nu va ceda procesorul decât atunci când a terminat ce avea de făcut "pe moment". Cu alte cuvinte, o parte din context (valorile regiștrilor) nu trebuie reținută pentru momentul în care task-ul va reintra în execuție. Schimbarea de context în acest caz se poate face foarte rapid, putând chiar să se facă salt direct la codul următorului fir.

Un dezavantaj mare al acestei metode este că buna funcționare a întregului sistem depinde de fiecare task, de cât de repede cedează procesorul. Întârzierile unui singur program duc la încetinirea întregului sistem.

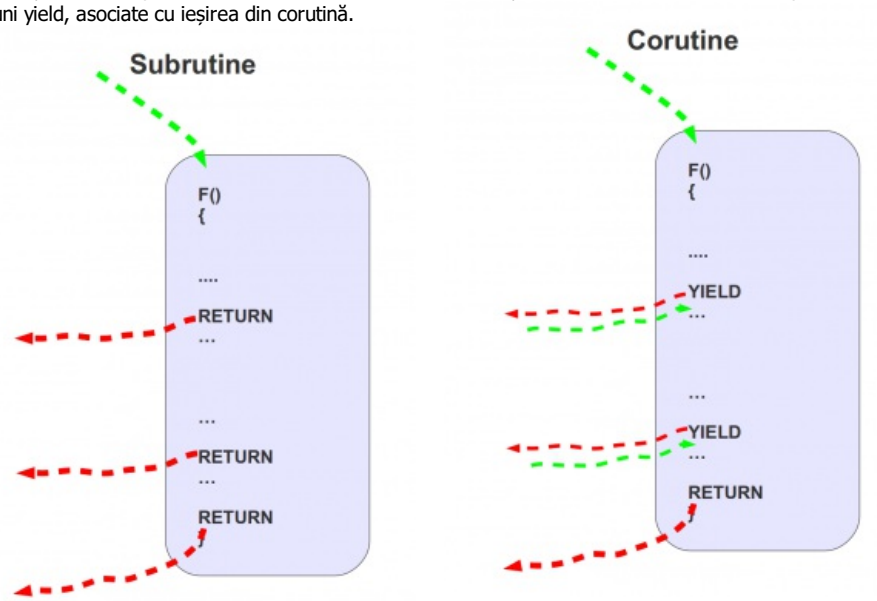
Principalul avantaj al unui sistem multi-tasking colaborativ care neglijează o parte din context este faptul că *nu necesită sincronizare*. Fiind un singur fir de execuție la un anumit timp, acesta sau va deține o resursă și o va folosi atât timp cât rulează, or nu o va deține.

Preemptiv

Multi-tasking-ul preemptiv este contrastant cu cel colaborativ, el promite fiecărui task o cuantă de timp de timp de rulare pe procesor. Odată ce un fir de execuție și-a terminat timpul alocat pe procesor el este preemptat și îi va lua locul un alt task care în nu și-a terminat cuanta de timp. Cuantele de timp sunt realocate periodic, fiecare task va avea șansa să termine ce a nu putut termina la expirarea cuantei precedente. Este important de reținut că în acest caz trebuie reținut întreg contextul execuției task-ului (regiștri, stivă), de exemplu o operație ADD pentru doi regiștri va avea nevoie de aceleași valori în cei doi regiștri ca înainte de preemptarea task-ului.

Corutine

Corutinele sunt structuri de cod ce conțin mai multe puncte de intrare și mai multe puncte de ieșire. Prin comparație, subrutinele sunt componente ce au un singur punct de intrare și mai multe puncte de ieșire. Primul punct de intrare al corutinelor este același cu cel al subrutinelor. Restul punctelor de intrare sunt locații care vin imediat după instrucțiuni yield, asociate cu ieșirea din corutină.



Duff's Device

Duff's Device este o implementare de copiere optimizată. Trucurile folosite sunt:

- case fără break: de la o clauză case se trece direct la următoarea
- case-urile nu au nevoie de cod structurat în blocuri în interiorul lor, deci putem avea un bloc cu mai multe case-uri. Case-urile se vor comporta ca label-uri la care se face jump în funcție de valoarea testată în switch.

Rezultatul este că se va face o copiere de câte 8 unități pentru fiecare iterație a buclei (mai puțin prima oară, când vor fi count % 8 copieri pentru a aduce numărul de unități care trebuiesc copiate la un număr divizibil cu 8. Copierea a câte 8 unități este mai eficientă pentru că elimină câte 7 comparații (comparațiile necesare la sfârșitul fiecărei iterații).

```
n = (count + 7) / 8
switch (count % 8) {
        case 0:         do {  *to = *from++;
        case 7:               *to = *from++;
        case 6:               *to = *from++;
        case 5:               *to = *from++;
        case 4:               *to = *from++;
        case 3:               *to = *from++;
        case 2:               *to = *from++;
```

```
        case 1:             *to = *from++;
                        } while (--n > 0);
    }
```

Corutine în C

Într-adevăr, se pot scrie corutine în C folosind mecanisme similare cu Duff's Device.

Se va folosi:

- O funcție normală
- O variabilă globală (sau definită local cu 'static') care să rețină starea curentă
- Alegerea punctului de intrare, în funcție de starea curentă

```
void f()
{
    static int state = 0;
    switch(state)
    {
    case 0:
        printf("prima rulare\n");
        state = 1;
        return;
    case 1:
        printf("a doua rulare\n");
        state = 2;
        return;
    case 2:
        printf("a treia rulare\n");
    }
}
```

Exerciții

1. Definiți macro-urile TASK_START(), TASK_YIELD() și TASK_END() astfel încât f() să poată fi rescrisă sub forma:

```
void f()
{
    TASK_START();
    printf("prima rulare\n");
    TASK_YIELD();
    printf("a doua rulare\n");
    TASK_YIELD();
    printf("a treia rulare\n");
    TASK_END();
}
```

HINT: Folosiți __LINE__ sau, dacă sunteți cu adevărat temerari, __label__.

2. Asigurați-vă ca macro-ul TASK_YIELD poate fi chemat de oricâte ori în cadrul funcției f 3. Fiind dată următoarea funcție f:

```
void f()
{
    static int i;
    TASK_START();
    for(i = 0; i < 3; i++)
    {
        printf("%d\n",i);
        TASK_YIELD();
    }
    TASK_END();
}
```

- Verificați că afișează numerele 0 1 2 atunci după trei apeluri
- De ce este folosit keyword-ul 'static' ?
- Scoateți keyword-ul 'static'. Ce observați?
- Puneți

  ```
  int i = 0
  ```

  în loc de declarația cu static. Ce observați? Care este explicația?

Contiki

În contextul rețelelor senzoriale wireless pe care urmează să le discutăm, vom folosi un sistem de operare pentru plăci cu microcontrollere AVR (nu AVR32!), și anume AVR Raven. Contiki este un sistem de operare construit în jurul a două stive de rețea pentru comunicație wireless, Rime și uIP (IPv6 ready). Este disponibil pe platformele Sky, T-Mote, MicaNode, AVR Raven, și altele.

Model de Multi-tasking în Contiki

Model de multitasking folosit în Contiki este cel de multitasking colaborativ, implementat cu corutine, ceea ce face programarea în el destul de anevoioasă. Fiecare program este constituit din mai multe procese, care sunt compilate și linkate static la executabilul final ce este pus pe nod (deci sunt cuplate strâns cu sistemul de operare).

Contiki thread

Un thread în Contiki este mapat pe o corutină, thread-ul este definit într-o singură funcție, pe mecanismul cu macro-uri pe care l-ați văzut deja. Mai multe thread-uri pot fi asociate cu același proces, unul este cel principal și pot fi mai multe care se ocupă de conexiuni.

Contiki process

- PROCESS THREAD (name, ev, data) - Definește corpul unui proces. Acest macro este folosit pentru a defini corpul (thread-ul principal al unui proces. Acest proces este chemat de fiecare dată când are loc un eveniment în sistem. Un proces începe întotdeauna cu PROCESS_BEGIN() și se încheie cu PROCESS_END()
- PROCESS_BEGIN() - Definește începutul unui proces
- PROCESS_END() - Definește sfârșitul unui proces
- PROCESS_YIELD() - Cedează procesorul
- PROCESS_WAIT_EVENT() - Așteaptă să fie postat un eveniment în sistem
- PROCESS_WAIT_EVENT_UNTIL© - Așteaptă să fie postat un eveniment în sistem, împreună cu o condiție.
- PROCESS_PAUSE() - cedează procesorul pentru un moment scurt de timp

Exerciții

1. Descărcați contiki-2.3
2. Dezarhivați și intrați în examples/hello_world/hello_world.c
3. Compilați exemplul cu
```
../examples/hello_world$ make TARGET=native
```
4. Rulați exemplul
```
../examples/hello_world$ ./hello_world.native
```
5. Adăugați PROCESS_PAUSE, împreună cu un printf după acesta. Ce observați?
6. Adăugați un timer:
   - struct etimer este structura necesară
   - etimer_set(&et, CLOCK_SECOND * 2) setează timer-ul să se declanșeze după 2 secunde
   - Cum așteptăm un eveniment?
   - etimer_expired(&et) verifică că a expirat într-adevăr timer-ul
7. Adăugați PROCESS_YIELD() și un printf după el. Ce observați?