WYKŁAD 3: Małe węzły dla systemów kontekstowych część II

Aleksander Pruszkowski
Instytut Telekomunikacji, Politechnika Warszawska

Przykładowe systemy operacyjne dla małych platform

TinyOS

- Historia i stan obecny
  - Twórcami TinyOS byli: David Culler i Kris Pister z UC Barkeley
  - Obecnie aktywnie działa ponad 1000 ośrodków naukowych wykorzystujących TinyOS
  - Obecna wersja TinyOS to: 2.1.x

TinyOS

- Czym jest TinyOS
  - System wykorzystujący odmianę języka „C” - NesC
    - Separuje implementację modułów od połączeń między modułami
  - Zbiór komponentów łączonych statycznie podczas kompilacji
    - Włączane są tylko używane fragmenty kodu
      - czyli to co potrzebne do działania aplikacji
      - nie ma typowego z systemów operacyjnych implementacji dynamicznego ładowania komponentów w czasie pracy
      - podejście zwiększa efektywność działania kodu
  - Zestawem narzędzi budowania aplikacji dla miniaturowych systemów o zasilaniu bateryjnym i łączności bezprzewodowej
  - Dość dobrze znany w kręgach akademickich systemem testowania nowatorskich rozwiązań

Idea i budowa TinyOS

- Tworzenie aplikacji - podstawy budowy
  - Oprogramowanie to zbiór komponentów
  - Konfiguracja - to opis jak je połączyć (przykład: BlinkAppC.nc)

```
configuration BlinkAppC{
}
implementation{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  ...

  BlinkC -> MainC.Boot;

  BlinkC.Timer0 -> Timer0;
  ...
  BlinkC.Leds -> LedsC;
}
```

Ta linia oznacza w notacji NesC:

```
BlinkC.Leds -> LedsC.Leds;
```

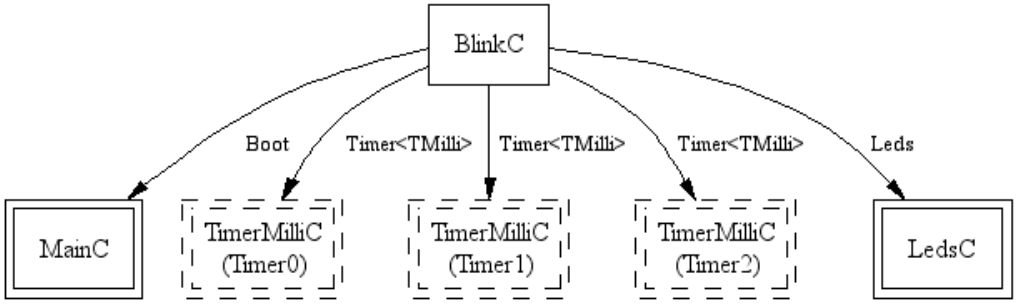
Idea i budowa TinyOS

- Tworzenie aplikacji - podstawy budowy, cd.
  - Moduł - kod implementacji komponentu (przykład: BlinkC.nc)

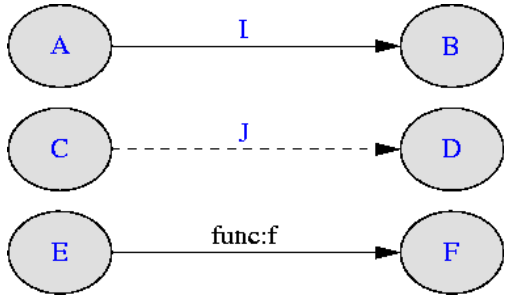
```
module BlinkC @safe(){
  uses interface Timer<TMilli> as Timer0;
  ...
  uses interface Leds;
  uses interface Boot;
}
implementation{
  event void Boot.booted() {
    call Timer0.startPeriodic( 250 );
  }
  event void Timer0.fired() {
    dbg("BlinkC", "Timer 0 fired @ %s.\n", sim_time_string());
    call Leds.led0Toggle();
  }
  ...
}
```

Idea i budowa TinyOS

- Tworzenie aplikacji - podstawy budowy, cd.
  - Powiązania między komponentami w aplikacji BlinkAppC



  - Ogólna koncepcja oznaczeń połączeń
    - Połączenie A z B
      A wymaga interfejsu I, który jest dostarczany przez B
    - Połączenie C z D
      C = D (równoważność)
    - Połączenie E z F
      E wymaga funkcji „f" a F dostarcza tę funkcję



Idea i budowa TinyOS

- Tworzenie aplikacji - podstawy budowy, cd.
  - Polecenia (ang. commands) i zdarzenia (ang. events)
    - Polecenia - podobieństwo do funkcji C
      - polecenia wydawane są innym komponentom (zgodnie z połączeniami zawartymi w pliku konfiguracji)
      - implementacja polecenie otrzymuje - w ramach wywołania - parametry
      - implementacja polecenia może zwracać wyniki po zakończeniu swego działania
        - np.: SUCCESS, FAIL
        - lub inne wartości zgodnie z deklaracją: `uint8_t`, `uint16_t`, `uint32_t`, ...

      przykładowy szkielet implementacji polecenia:

```
command result_t Send.send(TOS_MsgPtr pMsg) {
  ...
  return SUCCESS;
}
```

Idea i budowa TinyOS

- Tworzenie aplikacji - podstawy budowy, cd.
  - Zdarzenia
    - sposób komunikacji o zmianie stanu: sprzętu/usług/...

    np.:

```
...
myval = call MyHPL.ADCRead();
if (myval>mythreshold){
  signal MyInterface.alert(myval);
}
...
event void MyInterface.alert(int myval) {
  dbg("MyAPP", "Alert (val=%d)\n", myval);
  call Leds.led0Toggle();
}
```

Idea i budowa TinyOS

- Tworzenie aplikacji - podstawy budowy, cd.
  - Zdarzenia, cd.
    - obsługa zdarzeń - powinna być jak najkrótsza, jeżeli nie możliwe - dzieli się dwie fazy: pobrania danych (krótka) i przetworzenia danych (długa)

np.:

```
int myreading=0;
...
event void Timer0.fired() {
    myreading= call MyHPL.ADCRead();
    post processing();
}
```

Pobranie danych

Przekazanie do kolejki nowego zadania - przetwarzania danych

Idea i budowa TinyOS

- Tworzenie aplikacji - podstawy budowy, cd.
  - Zadania
    - Cel używania zadań
      - odłożenie czynności na później
      - uszeregowanie zadań w kolejności ich ważności
  - Przykładowy kod implementacji zadania

```
task void processing(){
   if (myreading>mythreshold)
       call Leds.redOn();
   else
       call Leds.redOff();
}
```

Idea i budowa TinyOS

- Tworzenie aplikacji - podstawy budowy, cd.
  - Oszczędność energii a działanie rozdzielnika (ang. schedulera)
    - Założenie: zadania są dla siebie „życzliwe” (kooperujące)
      - kooperatywnie podzielą się czasem procesora
    - Domyślnie: zadanie może zostać włożone do kolejki zadań tylko raz - potem czeka aż zostanie uruchomione
      - resztę takich prób tego samego zadania system zignoruje
      - ponowne włożenie do kolejki możliwe po przejęciu procesora przez zadanie
    - Kolejność w jakiej zostaną pobrane do wykonania zadania może być modyfikowana
      - możliwa zmiana implementacji rozdzielnika
    - Gdy system nie ma zadań do wykonania usypia węzeł (ang. sleep state)
      - główny sposób oszczędzania energii

Ograniczenia TinyOS

- Zalety i wady systemu
  - Wysoka oszczędność energetyczna - dla dłuższego działania na bateriach
    - Węzły mogą pracować miesiące/lata bez wymiany baterii
  - Niewielka moc obliczeniowa węzłów
    - Fosc np.: 100MHz - IntelMote2, 8MHz - MicaZ
  - System przeznaczony dla węzłów o małych zasobach pamięciowych
    - RAM: <8KB
    - ROM: <128KB

Contiki

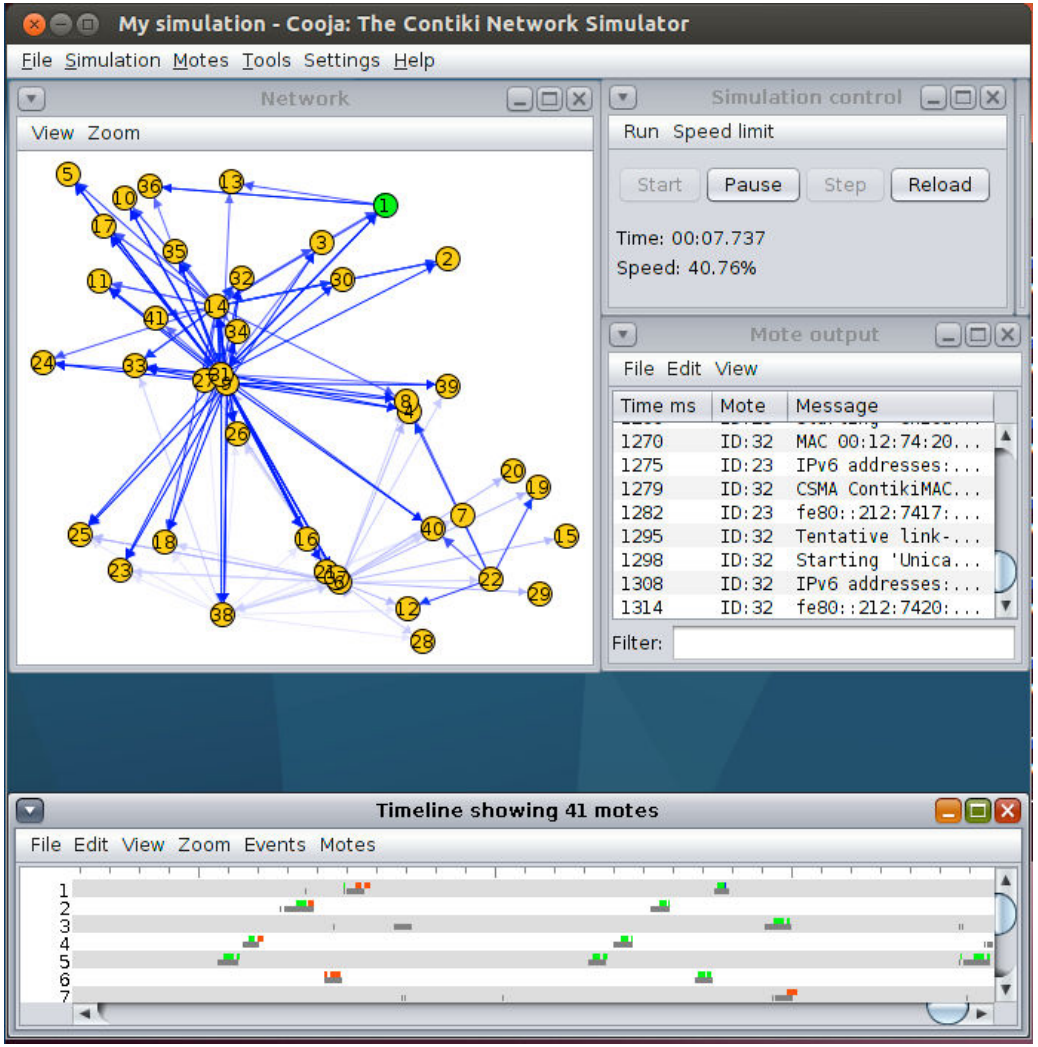
- System dla IoT („The Open Source OS for the Internet of Things”)
  - Przeznaczony dla małych, tanich i oszczędnych energetycznie procesorów
    - np.: zgodnych z 8051 SoC (Systems-On-a-Chip), MSP430, AVR, ARM
- Zapewnia możliwość podłączenia niemal każdego węzła do Internetu
  - Wsparcie dla: IPv4 i IPv6 oraz pełen sieciowy stos IP dla protokołów: UDP, TCP, HTTP, ...
  - Wsparcie dla nowych protokołów uwzględniających niskie pobory energii:
    - 6lowpan (IPv6 over Low power Wireless Personal Area Networks)
      - baza to IEEE 802.15.4 jako sposób dostarczania pakietów IP między węzłami
    - RPL (IPv6 **R**outing **P**rotocol for **L**LN, LLN=Low-Power and Lossy Networks)
      - protokół trasowania wspierający w małych urządzeniach modele ruchu typowe dla WSN: multipoint-to-central-point oraz central-point-to-multipoint oraz (z ograniczeniami) point-to-point
    - CoAP (**Co**nstrained **A**pplication **P**rotocol)
      - specjalistyczny protokół dla aplikacji M2M, zapewniający zgodność z HTTP dla małych urządzeń (np.: 8-bit MCU, mała ilość ROM/RAM, przepływność 10kbit/s)

Contiki

- Zasilanie bateryjne urządzeń
  - Założenie; „System ma działać lata na parze baterii AA”
    - Eliminacja luk (system/oprogramowanie/sprzęt) gdzie energia jest tracona
    - Własny protokół: ContikiMAC
      - zapewnia możliwość usypiania węzłów (w tym routerów) - dobre efekty osiągane dzięki dobranemu współczynnikowi „duty cycling”
- Kod systemu/aplikacji tworzony jest w standardowym C
  - Prostota przenoszenia systemu na nowe platformy
    - Łatwość tworzenia oprogramowania niskopoziomowego styku ze sprzętem w C
    - Odpada konieczność tworzenia trudnej do przeniesienia implementacji mechanizmu wywłaszczania
- System z otwartym kodem (Open Source)

Contiki

- Symulator Contiki (Cooja) - symulacja sieci radiowej urządzeń z uwzględnieniem topologii sieci
  - Symulacja realizowana przez kompilacje aplikacji dla natywnych platform i odpowiednie ich połączenie z symulatorem za pomocą JNI (Java Native Interfaces)
    - rozwiązanie uzyskuje lepsze emulowanie węzłów



Źródło: wikipedia.org

Contiki

- Inne cechy
  - Aktywne wsparcie społeczności akademickiej i przemysłowej
    - np.: Atmel, Cisco, ETH, Redwire LLC, SAP, Thingsquare, ...
  - Dynamiczne ładowanie modułów
    - Elementy oprogramowania można podmieniać podczas pracy
  - Niewielkie wymagane zasoby pamięciowe
    - Pełen stos IPv6 z „sleepy routers” wymaga mniej niż 30k ROM i 10k RAM
  - Własny system obsługi plików dla pamięci Flash węzłów
    - Coffee flash file system
      - lekka implementacja
      - zapewnia podstawowe operacje na plikach: open, close, read, write, append
      - zawiera mechanizmy pielęgnacji matrycy pamięci flash (flash wear-leveling problem)
      - wydajność systemu plików jest na poziomie 95% bezpośredniego dostępu do matrycy flash

Contiki

- Mechanizm „Protothreads”
  - Zapewnia tworzenie kodu wielowątkowego bez stosowania skomplikowanych niskopoziomowych mechanizmów wywłaszczania
    - Mechanizm wywłaszczania - przełącza procesor między zadaniami
      - mechanizm ma budowę skomplikowaną i mocno nie przenośną między procesorami
    - Typowa implementacja mechanizmu wywłaszczania wymaga dużych zasobów pamięciowych
      - „protothreads” ze swoim przełącznikiem zadowala się tylko 2...4B tej pamięci dla każdego wątku
  - „Protothreads” wykorzystują mechanizm pre-procesingu języka C dla zmiany kodu wielowątkowego w kod jedno wątkowy
  - Ważniejsze wady podejścia
    - Zmienne automatyczne nie są zachowywane przy przełączaniu zadań
      - aby nie stracić ich zawartości - trzeba je świadomie zachować przed przełączeniem się do innego wątku
      - lub zmienić ich charakter - np.: poprzez deklaracje takich zmiennych jako statyczne - unika się umieszczania takich zmiennych na stosie procesora

Contiki

- „Protothreads” API
  - PT_THREAD(name_args) - deklaracja implementacji wątku
  - PT_INIT(pt) - inicjalizacja wątku
  - PT_BEGIN()/PT_END() - deklaracja początku i końca właściwej sekcji wątku
  - PT_WAIT_UNTIL(pt, cond)/PT_WAIT_WHILE(pt, cond) - warunkowe blokowanie wątku
    - przydatne np.: gdy wątek czeka na dane z sensora
  - Dodatkowe funkcje
    - PT_WAIT_THREAD(pt, thread) - zaczekaj aż dziecko „tego” wątku zakończy swoje działanie
    - PT_SPAWN()/PT_RESTART()/PT_EXIT()/PT_SCHEDULE()/PT_YIELD()/PT_YIELD_UNTIL - rzadziej używane funkcje sterowania pracą wątków

Contiki

- Przykład (za źródłami w: esb\apps\beeper.c)

```
...
static struct pt beeper_pt;
static PT_THREAD(beeper_thread(struct pt *pt)){
  PT_BEGIN(pt);
  while(1){
    PT_WAIT_UNTIL(pt,
         etimer_expired(&etimer));
    etimer_reset(&etimer);
    leds_invert(LEDS_RED);
   ...
  }
  PT_END(pt);
}
```

```
void main(void){
    PT_INIT(&beeper_pt);
    ...
    while(1){
          beeper_thread(&beeper_pt);
          ...
    }
}
```

Contiki

- Jak działa mechanizm pre-procesingu:

```
1 int sender(pt) {
2 PT_BEGIN(pt);
3
4 /* ... */
5 do {
6
7        PT_WAIT_UNTIL(pt,
8                      cond1);
9
10 } while(cond);
11 /* ... */
12 PT_END(pt);
13
14 }
```

```
int sender(pt) {
   switch(pt->lc) {
   case 0:
     /* ... */
     do {
       pt->lc = 8;
   case 8:
       if(!cond1)
       return PT_WAITING;
     } while(cond);
     /* ... */
   }
   return PT_ENDED;
}
```

```
#define PT_BEGIN(pt)
#define PT_END(pt)
#define PT_WAIT_UNTIL(pt,c)
```

```
switch(pt->lc) { case 0:
} return PT_ENDED
pt->lc = __LINE__; \
case __LINE__: \
if(!(c)) return PT_WAITING
```

Źródło: dunkels06protothreads.pdf

Podsumowanie

Źródła uzupełniające

- Strony internetowe
  - Przykładowe systemy operacyjne dla małych platform
    - http://www.tinyos.net/
    - http://www.contiki-os.org/
- Adam Dunkels, „Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems”
    - http://en.wikipedia.org/wiki/Protothreads
    - http://en.wikipedia.org/wiki/Concurrent_computing

Dziękuję za uwagę!