Workshop: Tasks and synchronization

Arjen den Hamer

May 4, 2007

EMC (4K450)

Content

- Recap: Tasks in BrickOS
- Synchronization:

→ Semaphores

  - Critical sections
  - Task sequence

→ Wait event

Why multitasking

Multitasking makes your robot walk and chew gum at the same time.

Advantages:

- Each task can have its own priority and rate.
- Efficient CPU usage, time requirements
- Overview, each thread often has dedicated purpose

Disadvantages

- Communication needed, risks of dead-lock

$\rightarrow$ Synchronization

- – Task sequence: task B has to wait till task A finishes
- – Protection of critical sections (semaphores)

Recap

- BrickOS: Priority based preemptive scheduler combined with round-robin
- Scheduled every 20 ms
  → If running job is ready, task can voluntary yield CPU time: `yield`, `sleep`, `msleep`, `wait_event`, `sem_wait`
- Priorities:
  Main function: priority 10
  Highest priority: priority 20 → Don't use, task can not be killed.

Question: what can be seen as task?

Book: thread of execution that competes for CPU usage

Practically: part of the program with a particular purpose, priority and rate

Examples: control wheels, check for edges of platform

Recap II

`tid_t ThreadID_1;` initiate thread_ID

```
Thr_1 = execi( &thread_function, 0, NULL, 10, DEFAULT_STACK_SIZE );
```

- starts function `thread_function`
- 0 input parameters
- the pointer to the parameters is empty, i.e. `NULL`
- the process is running at priority 10

`kill(ThreadID_1);` kill process

Recap II

`tid_t ThreadID_1;` initiate thread_ID

```
Thr_1 = execi( &thread_function, 0, NULL, 10, DEFAULT_STACK_SIZE );
```

- starts function `thread_function`
- 0 input parameters
- the pointer to the parameters is empty, i.e. `NULL`
- the process is running at priority 10

`kill(ThreadID_1);` kill process

Easiest (not the most elegant) way to give variables to `thread_function` is via global variables.
→ Initiated variables outside main-function.

Synchronization: critical sections

Several tasks write on same source
→ Unpredictable results

```
#include <conio.h>
#include <unistd.h>

#define CNT_MAX 4000

unsigned int cnt; /* global variable */
int count(int cnt, char **argv);

int main(int argc, char **argv)
{
    tid_t Thr_1, Thr_2;
    cnt = 0;

    Thr_1 = execi( &count, 0, NULL, 15, DEFAULT_STACK_SIZE );
    Thr_2 = execi( &count, 0, NULL, 15, DEFAULT_STACK_SIZE );

    sleep(2);
    lcd_int(cnt);
    return 0;
}
```

Synchronization: critical sections II

```
int count(int argc, char **argv)
{
    int i;

    for (i=0;i<CNT_MAX;i++)
    {
        cnt++;
    }
    return 1;
}
```

Synchronization: critical sections II

```
int count(int argc, char **argv)
{
    int i;

    for (i=0;i<CNT_MAX;i++)
    {
        cnt++;
    }
    return 1;
}
```

result: $7235, 7585, \cdots \neq 8000$

What happens?

Machine code

three steps in machine code to increment counter:

```
LOAD R1,M(counter);      load value
INC R1;                  increment value
STORE R1,M(counter);     write value back
```

```
LOAD R1,M(counter);    counter = 0
INC R1;                counter = 0
```

context switch

```
LOAD R1,M(counter);    counter = 0
INC R1;                counter = 0
STORE R1;              counter = 1
```

context switch

```
STORE R1,M(counter);   counter = 1 ≠ 2!!
```

Protect critical section

Synchronization: using semaphores

```
#include <conio.h>
#include <unistd.h>
#include <semaphore.h>

#define CNT_MAX 4000

sem_t sem;

unsigned int cnt; /* global variable */
int count(int cnt, char **argv);

int main(int argc, char **argv)
{
    tid_t Thr_1, Thr_2;
    sem_init(&sem, 0, 1);

    cnt = 0;

    Thr_1 = execi( &count, 0, NULL, 15, DEFAULT_STACK_SIZE );
    Thr_2 = execi( &count, 0, NULL, 15, DEFAULT_STACK_SIZE );

    sleep(2);
    lcd_int(cnt);
    return 0;
}
```

Synchronization: using semaphores

```
int count(int argc, char **argv)
{
    int i;

    for (i=0;i<CNT_MAX;i++)
    {
        sem_wait(&sem);
        cnt++;
        sem_post(&sem);
    }
    return 1;
}
```

result: 8000

Synchronization: using semaphores

```
int count(int argc, char **argv)
{
    int i;

    for (i=0;i<CNT_MAX;i++)
    {
        sem_wait(&sem);
        cnt++;
        sem_post(&sem);
    }
    return 1;
}
```

result: 8000

Increase of semaphore also critical?
→ No: Atomic operation

Synchronization: semaphores

`sem_t sem;`	initialize semaphore
`sem_init(&sem, 0, 1);`	init semaphore to be one.
`sem_wait(&sem);`	wait till semaphore > 0, lower semaphore with one.
`sem_post(&sem);`	increment semaphore
`int sval;`	
⋮	
`sem_getvalue(&sem, &sval);`	checks current value of semaphore
`lcd_int(sval);`	

Synchronization: semaphores

`sem_t sem;`	initialize semaphore
`sem_init(&sem, 0, 1);`	init semaphore to be one.
`sem_wait(&sem);`	wait till semaphore > 0, lower semaphore with one.
`sem_post(&sem);`	increment semaphore

```
int sval;
⋮
sem_getvalue(&sem, &sval);
lcd_int(sval);
```

`sem_getvalue(&sem, &sval);` checks current value of semaphore

Big fat warning: If you choose to use pointers, allocate memory.

```
sem_t *psem;
psem = malloc(sizeof(sem_t));
sem_init(psem, 0, 1);
free(psem);
```

`free(psem);` Frees memory to prevent memory leakage

Synchronization: task sequence

Start of task B only makes sense if task A is finished.

→ init semaphores at zero: desired sequence A,B,C

pseudocode

A	B	C
⋮	`sem_wait(sem_1)`	`sem_wait(sem_2)`
⋮		
`sem_post(sem_1)`	⋮	
	`sem_post(sem_2)`	⋮

Synchronization with external signal

```
wait_event(event_function,0)
```

waits till function `event_function` returns 1.

```
wakeup_t event_function(wakeup_t not_used_here)
{
    return TRIGGER2;
}
```

Questions