4. Advanced Embedded Systems

- watch-dogs
- real-time
- multi-tasking

4.1 Watchdog

A watchdog is a specialized timer

- It is initialized to a certain value and keeps counting down
- If the watchdog counter reaches zero, an interrupt is raised
- A correctly running program will reset the watchdog timer in regular intervals to its initial value, so no interrupt will occur

Watchdog

```
int watchcount;

main()
{ watchreset();
  OSAttachTimer(1, watchdog);
  …
  while(1)
  { /* main processing loop */
    watchreset();
    …
  }
}
```

```
void watchdog() /* 100Hz */
{ watchcount--;
  if (watchcount<0) error("..");
}

void watchreset()
{ watchcount = 100;
}
```


Watchdog

- A watchdog is a very common and very effective tool for fault detection
- Can be used to detect hardware errors and software errors
- Especially useful for if a program "hangs"
- Interrupt or error routine called can reset system or restart individual task

4.2 Real-Time

- *Real-time systems* are sometimes confused with *high performance systems* — this is not the case
- A real-time system may be relatively slow, but it must be able to respond within a specified time frame
- Examples:
  - ABS running at 100Hz: each cycle of sensor data read, processing, actuator control must not last longer than 0.01s
  - Real-Time image processing PAL / full frames (50Hz/2): cycle time fir image capture, data transfer, processing, result transmission must be less than 40ms

Real-Time

Things get more complex with multi-tasking involved

- The sum of all all time slices must be less than the lowest cycle time of all tasks
- Multi-tasking with priorities can be used to improve system behavior
- An automated off-line tool for task scheduling or an on-line tool for monitoring task performance is required

4.3 Multitasking

- **Multithreading** is a simpler and faster version of **Multitasking**, which switches between parallel processes but keeps all processes in memory
- Several **processes** can be started in **parallel**. Since we only have **sequential hardware**, the operating system takes turns in executing each process
- Switching between processes (**time slicing**) is either initiated by the processes themselves (**cooperative**) or by the system timer (**preemptive**)
- Multithreading is a more general way of parallel processing than timer routines

Task Definition

- Task, Process, Thread
- Similar to function/ procedure no parameters, no return value
- Often contains (endless) loop
- Can read its own ID-no. `OSGetUID(0)`
- Can terminate itself `OSKill(0)`

```
void mytask()
{ int id, i;
  id = OSGetUID(0);  /* id */
  for (i=1; i<=100; i++)
    LCDPrintf("hello%d\n", id);
  OSKill(0);   /* terminate */
}
```

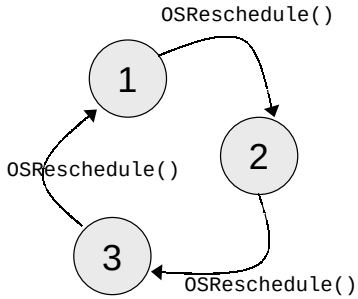
Task Start

- Task declaration as `struct tcb`
- Init multitasking as preemptive or cooper.
- Task spawning with name, code, stack size, priority, and id-no.
- Activating task `OSReady`
- Relinquish control `OSReschedule`

```
#define SSIZE  4096
struct tcb *task1, *task2;
…
OSMTInit(PREEMPT);  /* or coop */
task1 = OSSpawn("t1", task, SSIZE,MIN_PRI,1);
task2 = OSSpawn("t2", task, SSIZE,MIN_PRI,2);
if(!task1 || !task2) OSPanic(“err");
OSReady(task1); /* set state READY */
OSReady(task2);
OSPermit();     /* activate scheduler */
OSReschedule(); /* goto next task */
…
```


4.3.1 Cooperative Multitasking

- Each parallel thread decides itself when to relinquish control
- All threads must be *well behaved*
- `OSReschedule()`



Cooperative Multitasking

```
#include "eyebot.h"
#define SSIZE  4096
struct tcb *task1, *task2;

void mytask()
{ int id, i;
  id = OSGetUID(0); /* slave no. */
  for (i=1; i<=100; i++)
  { LCDPrintf("task %d : %d\n", id,i);
    OSReschedule(); /* trans. cntrl */
  }
  OSKill(0);  /* terminate thread */
}
```

```
int main()
{ OSMTInit(COOP);  /* init multitask. */
  task1 = OSSpawn("t1", mytask, SSIZE,
   MIN_PRI, 1);
  task2 = OSSpawn("t2", mytask, SSIZE,
   MIN_PRI, 2);
  if(!task1 || !task2) OSPanic(“err");

  OSReady(task1); /* set state READY */
  OSReady(task2);
  OSReschedule(); /* start multitask. */
  /* --------------------------- */
  /* go HERE when no READY thread */
  LCDPrintf("back to main");
  return 0;
};
```

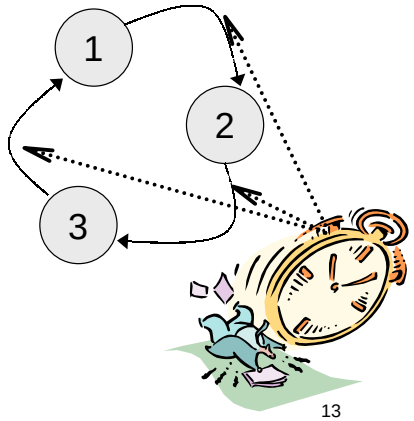

Cooperative Multitasking

Output cooperative tasks

```
task 2 : 1
task 1 : 1
task 2 : 2
task 1 : 2
task 2 : 3
task 1 : 3
...
task 2 : 100
task 1 : 100
back to main
```

4.3.2 Preemptive Multitasking

- External **scheduler** transfers control between tasks
- Use of time slices
- No need for `OSReschedule()`



Preemptive Multitasking

- Only differences in `main` are:
  `OSMTInit(PREEMPT);`
  …
  `OSPermit();`
- Task does not need:
  `OSReschedule()`
- *Is this all?*
  Potential problems?

```
void mytask()
{ int id, i;
  id = OSGetUID(0); /* slave no. */
  for (i=1; i<=100; i++)
  { LCDPrintf("task %d : %d\n", id,i);
  }
  OSKill(0);  /* terminate thread */
}
```


Preemptive Multitasking

Output preemptive tasks

```
task 1 : 1
task 1 : 2
tastask 2 : 1
task 2 : 2
task 2 :k 1: 3
task 1 : 4
...
```


4.3.3 Synchronization

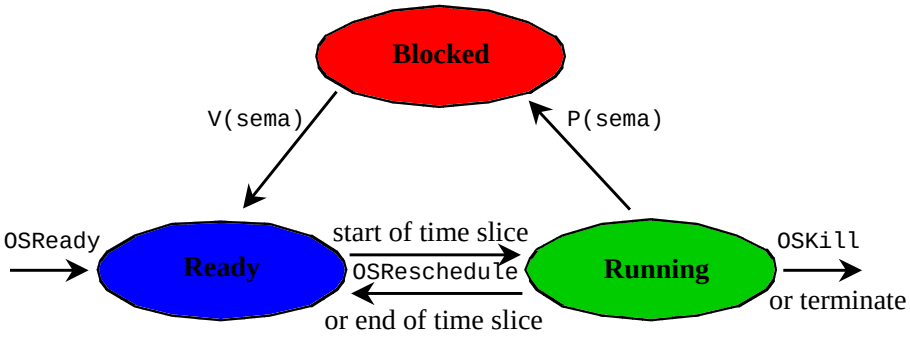
Problem:

- There is no way of telling when the time slice is up and the next task gets activated
- Task switch might come at a bad time
  e.g. here in middle of printout

Solution:

- **Task synchronization**
  - Semaphores ←
  - Monitors
  - Message passing

Task Model



Semaphores

- Semaphore is data structure for synchronization, with:
  - counter
  - list of blocked processes
- Can be free (1) or blocked (0)
- Binary semaphores vs. counting semaphores
- Block operation P (**p**ass) `OSSemP`
- Free operation V (lea**v**e) `OSSemV`

```
struct sem  my_sema;
OSSemInit(&my_sema, 1);
…
OSSemP(&my_sema);
 /* exclusive block, for
   example write to screen */
OSSemV(&my_sema);
```


Semaphores

- If semaphore is free (>0)
  - `OSSemP` will decrement its count by 1
  - `OSSemV` will increment its count by 1
- If semaphore is blocked (≤0)
  - `OSSemP` will decrement its count by 1 *and* **block** calling task
  - `OSSemV` will increment its count by 1 *and:* **free** next blocked task

Example Program

```
struct tcb *p1, *p2;
struct sem lcd;

int main()
{ OSMTInit(PREEMPT);  /* set multitasking mode */
  OSSemInit(&lcd,1);  /* Init semaphore: allow 1 writer */

  /* Init and ready threads */
  p1 = OSSpawn("P1", myproc, SSIZE, MIN_PRI, 1);
  p2 = OSSpawn("P2", myproc, SSIZE, MIN_PRI, 2);
  OSReady(p1);
  OSReady(p2);

  /* Start multitasking */
  OSPermit();     /* activate multitasking */
  OSReschedule(); /* startup time */

/* ------------------------------------------------------------------- */
/* processing will return HERE, when there is no READY thread left */
  return 0;
}
```

Semaphores

Sample Thread - *without Semaphore*

```
void myproc()
{ int id;

  id = OSGetUID(0);  /* read thread-no. supplied by main during spawn */

  while(1)
  {
      LCDPrintf(“thread %d\n", id);

  }
}
```


Semaphores

Sample Thread - *with Semaphore*

```
void myproc()
{ int id;

  id = OSGetUID(0);  /* read thread-no. supplied by main during spawn */

  while(1)
  { OSSemP(&lcd);
      LCDPrintf("thread %d\n", id);
    OSSemV(&lcd);
  }
}
```


Reader-Writer Example

1st try

```
/* global */
int stack[1000];
int sindex = -1;
```

```
void writer()
{ int w;

  for (w=0; w<1000; w++)
  {
      sindex++;
      stack[sindex] = w;

  }
}
```

```
void reader()
{ int r;

  for (r=0; r<1000; r++)
  {
      LCDPrintf(“%d ”, stack[sindex]);
      sindex--;

  }
}
```


Reader-Writer Example

2nd try

```
/* global */
int stack[1000];
int sindex = -1;
struct sema buf;
```

```
void writer()
{ int w;

  for (w=0; w<1000; w++)
  { OSSemP(&buf);
      sindex++;
      stack[sindex] = w;
    OSSemV(&buf);
  }
}
```

```
void reader()
{ int r;

  for (r=0; r<1000; r++)
  { OSSemP(&buf);
      LCDPrintf(“%d ”, stack[sindex]);
      sindex--;
    OSSemV(&buf);
  }
}
```

Reader-Writer Example

3rd try

```
/* global */
int stack[1000];
int sindex = -1;
struct sema buf;
```

```
void writer()
{ int w;

  for (w=0; w<1000; w++)
  { OSSemP(&buf);
      sindex++;
      stack[sindex] = w;
    OSSemV(&buf);
  }
}
```

```
void reader()
{ int r =0;

  while (r<1000)
  { OSSemP(&buf);
    if (sindex>=0)
    { LCDPrintf("%d ", stack[sindex]);
      sindex--;
      r++;
    }
    OSSemV(&buf);
  }
}
```


Reader-Writer Example

- Still not perfect (busy wait!)
- Better use additional semaphores for empty/full
  - Semaphore for mutual exclusion (buf)
  - Semaphore for full
  - Semaphore for empty
- ⇒ Bounded-Buffer problem

4.3.4 Scheduling

- Scheduling refers to the execution order of tasks (using time slices), as determined by an operating system component ("scheduler")
- Scheduling can be
  - without priorities
  - with static priority
  - with dynamic priority

Scheduling - No Priorities

- All tasks treated equal
- "round robin" principle
- Sequence: t_1, t_2, t_3, t_1, t_2, t_3, t_1, t_2, t_3, …

Active	Ready-Queue
t_1	$[t_2, t_3]$
t_2	$[t_3, t_1]$
t_3	$[t_1, t_2]$
…	

Scheduling - No Priorities

Assume a task gets blocked:

t_1 (*block* t_1) t_2, t_3, t_2, t_3 (*unblock* t_1) $t_2, t_1, t_3, t_2, t_1, t_3, \ldots$

Active	Ready	Blocked	
t_1	$[t_2, t_3]$	{}	→ *t_1 is being blocked*
t_2	$[t_3]$	$\{t_1\}$	
t_3	$[t_2]$	$\{t_1\}$	
t_2	$[t_3]$	$\{t_1\}$	
t_3	$[t_2, t_1]$	{}	→ *t_3 unblocks t_1*
t_2	$[t_1, t_3]$	{}	
t_1	$[t_3, t_2]$	{}	
t_3	$[t_2, t_1]$	{}	
...			

Scheduling - Static Priorities

- Each task is started with an integer priorities
- Tasks with higher priorities should be given preferred treatment compared to tasks with lower priority
- For static priority, this means that a task with priority p can only then run when no task with priority p+1 or higher is ready.

Scheduling - Static Priorities

Example: t_A, t_B (priority 2), t_a, t_b (priority 1)

Active	Ready	Blocked	
t_A	$[t_B, t_a, t_b]$	{}	
t_B	$[t_A, t_a, t_b]$	{}	
t_A	$[t_B, t_a, t_b]$	{}	→ *t_A blocked*
t_B	$[t_A, t_a, t_b]$	$\{t_A\}$	→ *t_B blocked*
t_a	$[t_b]$	$\{t_A, t_B\}$	
...			

Scheduling - Static Priorities

Problem with static priorities:

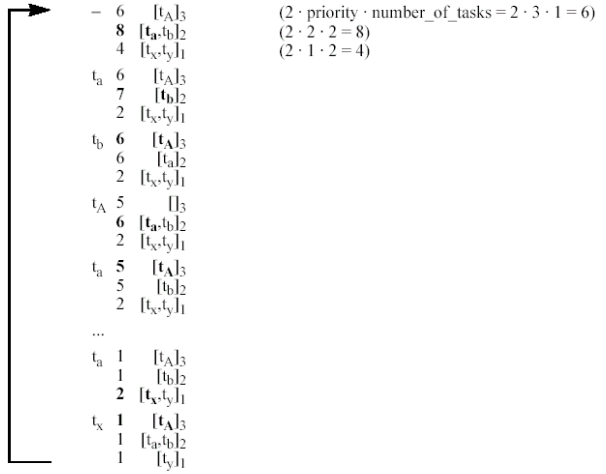
- "Starvation"
  A lower priority task may never be executed while higher priority tasks are using the CPU

→ *Dynamic Priorities*

Scheduling - Dynamic Priorities

- **Priority queues** are maintained for each static priority value (e.g. 1..8)
- Queues have dynamic values that are initialized with:
  2 * static_value * queue_elements
- Every time a task from a queue is executed, the dynamic queue value is decremented by 1
- If a dynamic queue value becomes 0,
  it will be reset to the initial value
- Scheduler always selects next ready task from queue with highest dynamic value

Scheduling - Dynamic Priorities



Multitasking Operations

```
int OSMTInit(BYTE mode);
        Input:          (mode) operation mode
                        Valid values are: COOP=DEFAULT,PREEMPT
        Output:         NONE
        Semantics:      Initialize multithreading environment

tcb *OSSpawn(char *name,int code,int stksiz,int pri,int uid);
        Input:          (name) pointer to thread name
                        (code) thread start address
                        (stksize) size of thread stack
                        (pri) thread priority
                        Valid values are: MINPRI-MAXPRI
                        (uid) thread user id
        Output:         (returncode) pointer to initialized thread
                        control block
        Semantics:      Initialize new thread, tcb is initialized and
                        inserted in scheduler queue but not set to
                        READY

int OSReady(struct tcb *thread);
        Input:          (thread) pointer to thread control block
        Output:         NONE
        Semantics:      Set status of given thread to READY
```


Multitasking Operations

```
int OSReschedule(void);
        Input:          NONE
        Output:         NONE
        Semantics:      Choose new current thread

int OSGetUID(thread);
        Input:          (thread) pointer to thread control block
                                (tcb *)0 for current thread
        Output:         (returncode) UID of thread
        Semantics:      Get the UID of the given thread

int OSKill(struct tcb *thread);
        Input:          (thread) pointer to thread control block
        Output:         NONE
        Semantics:      Remove given thread and reschedule

int OSSleep(int n)
        Input:          (n) number of 1/100 secs to sleep
        Output:         NONE
        Semantics:      Let current thread sleep for at least n*1/100
                        seconds.  In multithreaded mode, this will
                        reschedule another thread.  Outside
                        multi-threaded mode, it will call OSWait().
```

Semaphore Operations

```
int OSSemInit(struct sem *sem,int val);
        Input:          (sem) pointer to a semaphore
                        (val) start value
        Output:         NONE
        Semantics:      Initialize semaphore with given start value

int OSSemP(struct sem *sem);
        Input:          (sem) pointer to a semaphore
        Output:         NONE
        Semantics:      Do semaphore P (down) operation

int OSSemV(struct sem *sem);
        Input:          (sem) pointer to a semaphore
        Output:         NONE
        Semantics:      Do semaphore V (up) operation
```