...

- Interrupts as “poor mans” multitasking
- Embedded systems – realtime kernels
  - krnl, (FreeRTOS,…)
- For small embedded systems
  - No Memory management/protection == no protection against “wild pointers”
  - aka no mercy systems !!!
- Task concept
  - Multitasking: threads, tasks,…
  - Using functions as body code for task
  - Queues for dynamic handling tasks exectioon(activeQ, sleepQ,…)
    - running ↔ ready ↔ sleeping
  - (Static) task priority for selection of who to run
    - Just a number”
- createtask, deleteTask, changeTaskPriority,… you name it :-)

- Left for the reader – digging into POSIX threads/multitasking
  - (https://computing.llnl.gov/tutorials/pthreads/)

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0; t<NUM_THREADS; t++){
      printf("In main: creating thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

- Semaphores – a synchronization object (invented 1962/1963)
  - (https://en.wikipedia.org/wiki/Semaphore_(programming))

  Synchronization between
  - Two tasks
  - ISR and task
  - Critical region protection
    - aka Mutex
- Considered as the most simple and most used primitive in emb sys
- Semaphores is often used as internal basic building block for other primitives.
- create semaphore
- wait on semaphore
  - tasks only a
- signal to a semaphore
  - tasks and ISRs
- attach a cyclic timer to a semaphore for later repeated waitin
- and more adv features not touched in the course (also for POSIX)

```
39 void t11(void) {
40
41   while (1) {
42     k_wait(sem1,0); // wait forever
43     doBlink();
44   }
45 }
46
47 void t22(void) {
48
49   while (1) {
50     k_sleep(100);
51     k_signal(sem1); // kick a signal to who cares (do not know receiver)
52     doBlink();
53   }
54 }
```

- t11 starts waiting
- t22 starts signaling
- sem1 and sem2 are both initialized to “1” - meaning on “key” present

- Why not same order of wait/signal inn both tasks ???

```
39 void t11(void) {
40
41   while (1) {
42     k_wait(sem1,0); // wait forever
43     k_signal(sem2);
44     doBlink();
45   }
46 }
47
48 void t22(void) {
49
50   while (1) {
51     k_sleep(100);
52     k_signal(sem1); // kick a signal to who cares (do not know receiver)
53     k_wait(sem2,0);
54     doBlink();
55   }
56 }
```

- Between tasks
- Between(from) ISR to task
- Synchronous (aka rendevouz)
- Asynchronous (buffer in between sender and r

```
39 void t1(void) {
40
41   while (1) {
42     k_wait(sem1,0); // wait forever
43     doBlink();
44   }
45
46 }
47
48 //  krnlisrsemkick.ino
49  ISR(INT0_vect, ISR_NAKED) {
50
51   // no local vars ?!?  ! I think
52   PUSHREGS();
53   if (!k_running)
54     goto exitt ;
55
56   icnt++;
57   ki_signal(sem1);
58   K_CHG_STAK();
59 exitt:
60
61   POPREGS();
62   RETI();
63 }
```

- Buffered coordinated dataflow from one task to another

```
18
19 struct k_msg_t   *pMsg2;
20 char mar2[10 * 2];  // 10 ints each 2 bytes   could instead write 10 *sizeof(int)
21
22 void t1(void) {
23   int i;
24   while (1) {
25
26     if (0 <= k_receive(pMsg2, &i, -1, NULL) ) {
27       doBlink();
28     }
29   }
30 }
31
32 void t2(void) {
33   int i;
34   i = 0;
35   while (1) {
36     k_sleep(20); // just ZZZZZZZZZZZZZZZ
37     k_send(pMsg2, &i); //just send 0,1,2,3,4....
38     i++;
39   }
40 }
41
42 void setup() {
43
44   Serial.begin(9600);
45
46   k_init(2, 0, 1); // from now you can crt task,sem etc
47   p_t1 = k_crt_task(t1, 10, s1, STK_SIZE);
48   p_t2 = k_crt_task(t2, 9, s2, STK_SIZE);
49
50   pMsg2 = k_crt_send_Q(10, 2, mar2);
51   k_start(10); // now we are runnning   with timer 10 msev
52 }
```

- All msg's of same size
- You decide
- You supply with memory
- You can
  - wait forever
  - timeout
  - have buffer filled
  - if no space for new msg
  - its just dropped !
- Messages are copied to send sys internal buffer
- you did supply in crt call
- So NO dynamic sh*t

- Same mechanism

```
struct k_t * p_t1;

struct k_msg_t *pMsg;
char mar[10 * 2];


void t1(void) {
  int i;
  while (1) {

    if (0 <= k_receive(pMsg, &i, 0, NULL) ) {
      doBlink();
    }
  }
}

ISR(INT0_vect, ISR_NAKED) { // UNO only
  static int icnt = 0;
  PUSHREGS();
  if (!k_running)     goto exitt ;

  icnt++;
  ki_send(pMsg, (void *)&icnt);
  K_CHG_STAK();

exitt:
  POPREGS();
  RETI();
}



void setup() {
  Serial.begin(9600);

  k_init(1, 0, 1); // from now you can crt task,sem etc
  p_t1 = k_crt_task(t1, 10, s1, STK_SIZE);
  pMsg = k_crt_send_Q(10, 2, mar);
  installISR (); // left out here
  k_start(10); // now we are runnning   with timer 10 msev
}
```

- signal to another task from ISR and task
- receive signals
- do mutexes for tasks
- Send information from task or ISR to task
  - fixed message size
- Receive information in tasks

- Message system internally
  - A ringbuffer applied on the data/array you supply with in crt_send call

- You cant !!
  - change ringbuffer size after k_start
  - change size of items in ringbuffer after k_start

- WHY ?? (this goes for all primitives)

- Because system must not change behaviour/metrics during execution

- Producing task/ISR is normallyhigh priority
- Receiving task may(may!) eat messages in chunks to
- task1/ISR p p pppp p p p pp
- task2 r r r(and waiting maybe)s

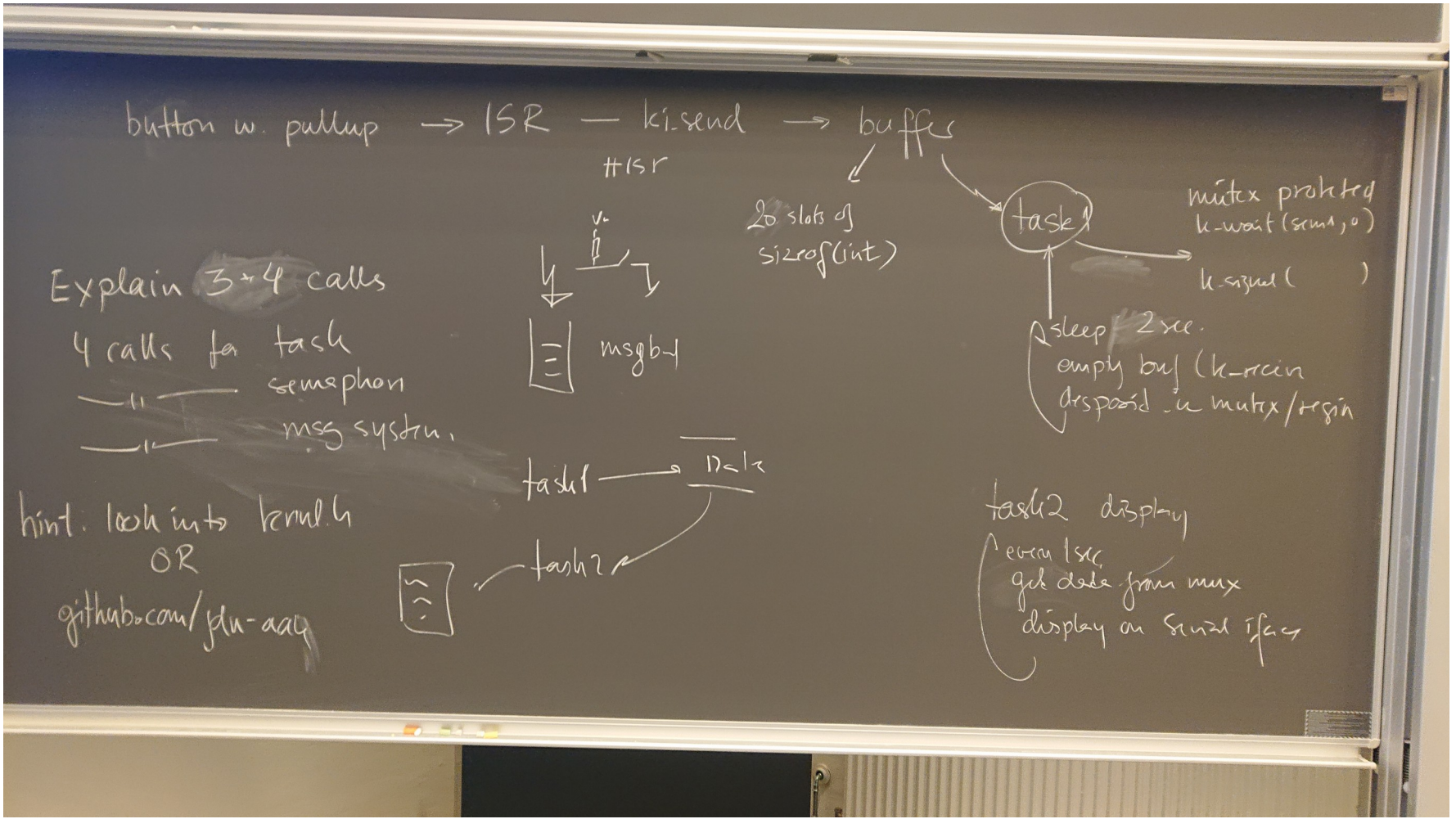
- You might find libraries where “read” form an external device is busy waiting on until some timeout

- should be coded with interfaces based on
  - task waiting for data is temporarily blocked by a semaphore

1) ISR Semaphore kick
   - ISR triggered by button
   - signaling to semaphore
   - Task receiving signal and printing kick
- ISR Message kick
   - As 1) except an int counting nr of interrupts is received by use of msgQ

- Design a debouncing mechanism in ISRs (as first lecture)
- Add counter so we can
  see how many
  bounces
  Send nr of bunces back
  by ki_send and …
  reset counter
-

```
32 ISR(INT0_vect, ISR_NAKED) { // UNO only
33   static int unsigned long t = 0;
34   PUSHREGS();
35   if (!k_running)     goto exitt ;
36
37   if ( (millis() - t) > 500) {
38     t = millis();
39     icnt++;
40     ki_send(pMsg, (void *)&icnt);
41
42   }
43   K_CHG_STAK();
44
45 exitt:
46   POPREGS();
47   RETI();
48 }
```


button w. pullup → ISR — ki_send → buffer
#isr
20 slots of sizeof(int)
task1
mutex protected
k_wait(sem1,0)
k_signal( )
sleep 2 sec.
empty buf (k_rcein
dispatch in mutex/region
msgbuf
Explain 3-4 calls
4 calls for task
semaphor
msg system.
hint. look into kernel.h
OR
github.com/glu-aau
task1 → IDLE
task2
task2 display
every 1sec
get data from mux
display on Serial iface