# Lecture 16: Interprocess Communication, Critical Sections

Vikram Iyer

Adapted from material by Blake Hannaford

# Announcements

Lab 3- 1 day extension

- Friday = 1 late days,
  Sunday (weekend) = 2 late days

Lab 4 will be up soon

- Due during finals week to give you max time/flexibility to work on it

Quiz on Wed 5/29 during lecture (20 min)

- In class, written, multiple choice (15 pts)
- Goal is to test concepts introduced after the midterm
- List of concepts will be posted by Friday

| Assignment | Points |
| --- | --- |
| C prog 1 | 20 |
| C prog 2 | 20 |
| Quiz | 15 |
| Lab 1 | 70 |
| Lab 2 | 70 |
| Lab 3 | 70 |
| Lab 4 / Project | 90 |
| Midterm | 45 |
| No Final | |
| **TOTAL** | **400** |

# Last time: FreeRTOS examples

blink_analogRead474_queue | Arduino 1.8.19

blink_analogRead474_queue

```
void TaskBlink2(void *pvParameters)  // This is a task.
{
   // initialize digital LED_BUILTIN on pin 13 as an output.
  pinMode(LED_PIN2, OUTPUT);
  int offTime = 500;

  for (;;) // A Task shall never return or exit.
  {
    xQueueReceive(Q_delay, &offTime, 0);
    Serial.println(offTime);
    digitalWrite(LED_PIN2, HIGH);   // turn the LED on (HIGH is the voltage level)
    vTaskDelay( 500 / portTICK_PERIOD_MS ); // wait for one second
    digitalWrite(LED_PIN2, LOW);    // turn the LED off by making the voltage LOW
    vTaskDelay( offTime / portTICK_PERIOD_MS ); // wait for one second
  }
}

void TaskAnalogRead(void *pvParameters)  // This is a task.
{
  for (;;)
```

88 Arduino Mega or Mega 2560, ATmega2560 (Mega 2560) on /dev/cu.usbmodem21401

# Producer-Consumer Model

**Producer:** A task which generates blocks of data.

**Consumer:** A task which does something with the data blocks and discards them or passes them to another consumer.

**Human instructions: Producer**

```
If there is room
  add item to shared buffers
else
  wait;
```

**Human instructions: Consumer**

```
If there are items in buffer
  process items
else
  wait;
```

# Producer-Consumer Pseudocode

```
/* example.h   */
#define BSIZE = 5
typedef item {plastic button};
struct item ConsIt, NextIt, buffer[BSIZE];
int in=0, out=0, count=0;
```

**Producer:**

```
while(1) {
  while(count >= BSIZE);
  count++;
  NextIt = {produce an item};
  buffer[in] = NextIt;
  in++;
  if(in >= BSIZE) in = 0;
}
```

**Consumer:**

```
while(1) {
  while(count==0); //"spinlock"
  count--;
  ConsIt = buffer[out]; out++;
  if(out >= BSIZE)
  out = 0;
  Take(ConsIt); //Consume item
}
```

# Intertask Communication and Data Sharing

**Method 1: Shared memory**

- Global Variables
- Shared Buffer
- Ring Buffer
- FIFO

# Shared Buffer(s)

- Producer fills a buffer
- Signals Consumer
- Consumer clears buffer

**Multiple buffers:** Producer fills buffer A while Consumer clears buffer B.
switch pointers A and B

This scheme is called **"double buffering"**. Allows producer and consumer to work simultaneously.

# Ring Buffer/FIFO

One buffer. Two Pointers *in, *out

**Producer**

```
while(in != out) {
  buffer[in++] = {new data}
  if(in >= BUFFER_SIZE)
    in = BUFFER;
}
print "BUFFER OVERFLOW!!!!" ;
halt;
```

**Consumer**

```
while(out != in ) {
  data = buffer[out++];
  if(out >= BUFFER_END)
    out = BUFFER;
}
print "BUFFER UNDERFLOW!!!!" ;
halt;
```

# LIFO

"Last-In-First-Out" A stack
One buffer, One pointer *iop

**Producer**

```
while(iop <= BUFFER_END) {
  buffer[iop++] = {new data}
}
print "LIFO OVERFLOW!!!!" ;
halt;
```

**Consumer**

```
while(iop != BUFFER ) {
  data = buffer[--iop];
}
print "LIFO UNDERFLOW!!!!" ; halt;
```

# Intertask Communication and Data Sharing

**Method 1: Shared memory**

- Global Variables
- Shared Buffer
- Ring Buffer
- FIFO

**Method 2: Message Passing**

Problem with shared memory approaches is that processes are not protected from each others' bugs. OS can isolate processes better by supporting messages.

**Two types:**

- Message Passing
- Mailbox

# Passing Data: Message passing

```
/* Process 1*/
while(1) {
  compute & produce data;
  OS_send_message(Proc_ID);
}
```

# Passing Data: Mailboxes

| | |
|---|---|
| `/* Process 1 */`<br>`name="mailbox1_2";`<br>`status = mailbox_setup(name);`<br>`if(status != MB_SUCCESS)`<br>`  error_exit(”Setup error");`<br>`while(1) {`<br>`  compute & produce data;`<br>`  OS_post_message(name);`<br>`}` | `/* Process 2 */`<br>`name ="mailbox1_2";`<br>`// assume Proc1 set up mailbox`<br><br>`while(1) {`<br>`  OS_pend_message(name);`<br>`  consume data;`<br>`}` |

OS message calls invoke the scheduler.

**Sender** is set to WAITING until receiver gets the message.

**Receiver** is set to WAITING until mailbox contains a message.

# Concurency Problem Statement

Fundamental Problem: make sure only one task accesses a resource during “Critical Section”.

**Critical Section:** a short segment of code which must be done as a unit (also called “atomic” operation).

**Above example:** testing buffer counter and taking/putting token must be done together without interruption.

# Fix 1: Global interrupt mask

Make an OS call, or manipulate bits in interrupt controller to prevent any ISRs from running during Critical Section.

```
OS_INT_MASK();// block interrupts

if(data_avail_flag) {
  // if ISR occurs here --->
  //Trouble!!
  process(buffer);
  data_avail_flag = FALSE;
  buffer_avail_flag = TRUE;
}
OS_INT_ENABLE();
```

This prevents:

1) pre-emption by scheduler which would start another task.

2) An ISR which may mess with buffers/ptrs.

**Pros:** Fast

**Cons:** Too greedy: blocks all I/O devices and other tasks whether they use the key resource or not!

# Fix 2: Test and set instruction

Use a specific machine language instruction that tests and sets a value in 1 cycle. For example some architectures have `TSR` a single instruction and cannot be interrupted.

```
# assembler:
  TSR R, FailAddr # jump to FailAddr if R ne 0
# explanation:
  if(R==0) R = 1 ;
  else jump FailAddr ;
```

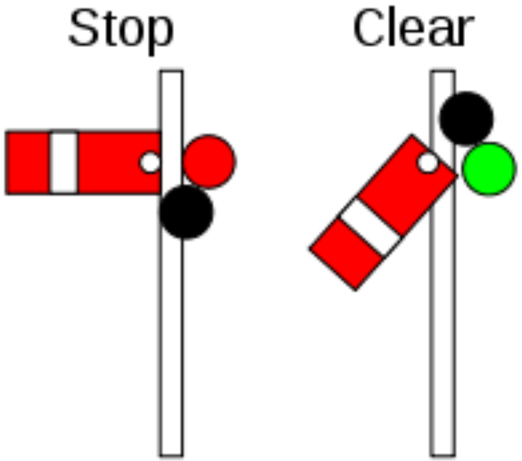
**Pros:**

Extremely fast and can be specialized for many resources without blocking unnecessarily.

**Cons:**

Very low-level. Only allows one user per resource. What if resource permits N users?

# Semaphores




Term Origin: Flags used in signaling.

**Semaphore**: An OS facility which guarantees exclusive access.

Three OS Calls:

```
semaphore os_get_semaphore(int N);
     // establish a new semaphore
     // initialize to N

void os_pend(semaphore S);
     // wait for resource protected by S

void os_post(semaphore S);
     // free up resource protected by S
```

# Semaphores: Pend and post

Identify the critical section in your code. "protect" it with `Pend(S)` and `Post(S)`:

```
...
pend(S);

critical section

code post(S)
```

# Implementing Pend and Post inside a kernel

Notes:

S is typically a small integer.

S==0 → wait, S ≥ 1 → go

```
Pend (S):
  Disable interrupts
  if (S>0) {
    S--;
    EnableInterrupts;
    return
  }
  else {
    Enable Interrupts;
    set task to 'WAIT';
    Store in TCB.sem; start scheduler
  }
```

```
Post(S):
  S++
  start scheduler
```

# Critical Section Summary

The **Contention Problem** can occur under two conditions:

1. There is preemption due to interrupts.
2. Two or more processes share a single resource.

A **Critical Section** is the part of code which, if interrupted, could cause a bug in sharing a resource between two processes.

Example: Increment a counter and then put data in buffer.

**Semaphores** can be used to protect a critical section.

1. One semaphore, S, is established for each shared resource.
2. pend(S) operation is used at start of critical section.
3. post(S) operation is used at end of critical section. Pend means "wait" (i.e. "patent pending").

Scheduler makes sure that tasks waiting for a resource (i.e.  pending a semaphore) are set to "WAITING" and that other tasks can run instead

# Semaphores: Pros and Cons

**Pros:**

- Specialized, one semaphore per resource, no unnecessary blocking.
- Widely used standard.
- Low to medium implementation overhead in O/S.

**Cons:**

- Can cause priority inversion

# Priority Inversion

```
Task  A  Priority    5 LOW
Task  B  Priority   10 MED
Task  C  Priority  15 HIGH
```

**Consider the following scenario**

Task A, C → Buffer → `semaphore S`

1. Task C is running and uses OS `pend(S)` to get the buffer. OS grants the buffer to C and C computes slowly on buffer.
2. Task A starts. Task A also requests buffer by OS `pend(S)`. It has to wait for C to finish with S.
3. With Task A still waiting, Task B starts.
4. C is pre-empted by an interrupt.
5. Scheduler runs B. Although B does not even want the buffer (semaphore S), it has higher priority than A, and blocks A ...
6. C is effectively blocked by lower priority task B. (C is waiting for A to release semaphore(S).

# Priority Inversion Fixes

- **Priority Inheritance.** O/S has to check each semaphore pend.
  If a higher priority process is blocked by a pend, process having the resource temporarily gets priority equal to the blocked process.

- **Priority Ceiling Protocol.** When getting a resource, process temporarily gets priority equal to highest process sharing that resource.