

# Lecture 17: Interprocess Communication, Critical Sections

Vikram Iyer

# Announcements

- Lab 3 demo due today
- Pick up midterms in OH
- Lab 4 is up, due at end of finals week

# 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 is room
  add item to shared buffers
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

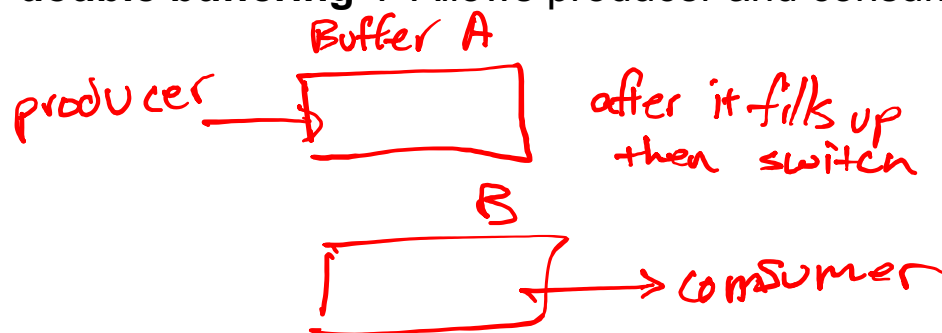
# Shared Buffer(s)

- 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;
```

*buffer is too full*

## Consumer

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

*trying to read but  
there's nothing there*

# 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 */
name="mailbox1_2";
status = mailbox_setup(name);
if(status != MB_SUCCESS)
    error_exit("Setup error");
while(1) {
    compute & produce data;
    OS_post_message(name);
}
```

```
/* Process 2 */
name = "mailbox1_2";
// assume Procl set up mailbox

while(1) {
    OS_pend_message(name);
    consume data;
}
```

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.

# Concurrency 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

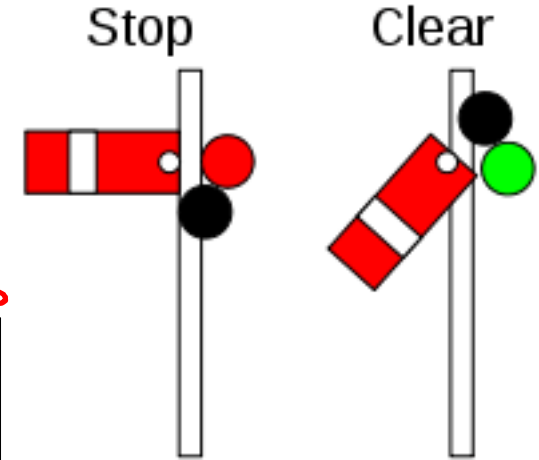
Term Origin: Flags used in signaling.

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

Three OS Calls:

*↙ number of processes*

```
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)
```

*wait for resource to be available*

*← tell the OS you're done*

# Implementing Pend and Post inside a kernel

Notes:

S is typically a small integer.

$S=0 \rightarrow$  wait,  $S \geq 1 \rightarrow$  go

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

*interrupts disabled*

```
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.  $\text{pend}(S)$  operation is used at start of critical section.
3.  $\text{post}(S)$  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

## Consider the following scenario

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

Task A, C  $\rightarrow$  Buffer  $\rightarrow$  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.

priority inversion

