Buffer and Timer Management

CPE 545

Chapter Summary

- A strategy for Buffer and Timer Management
- Buffers are used for the interchange of data among modules in a communications system.
- Timers are used for keeping track of:
  - Timeouts for messages to be sent,
  - Acknowledgements to be received,
  - Aging out of information in tables.

Buffer Management

- Mechanisms to create and operate buffers with a focus on reducing data copying operations
  - Allocation - Reserving buffer from a global buffer memory pool
  - Manipulation of buffers - All operations affecting data stored in the buffer.
    - Copying, deleting, updating, concatenating two buffers etc.
  - Freeing buffers - Returning a buffer to the global buffer pool for re-use

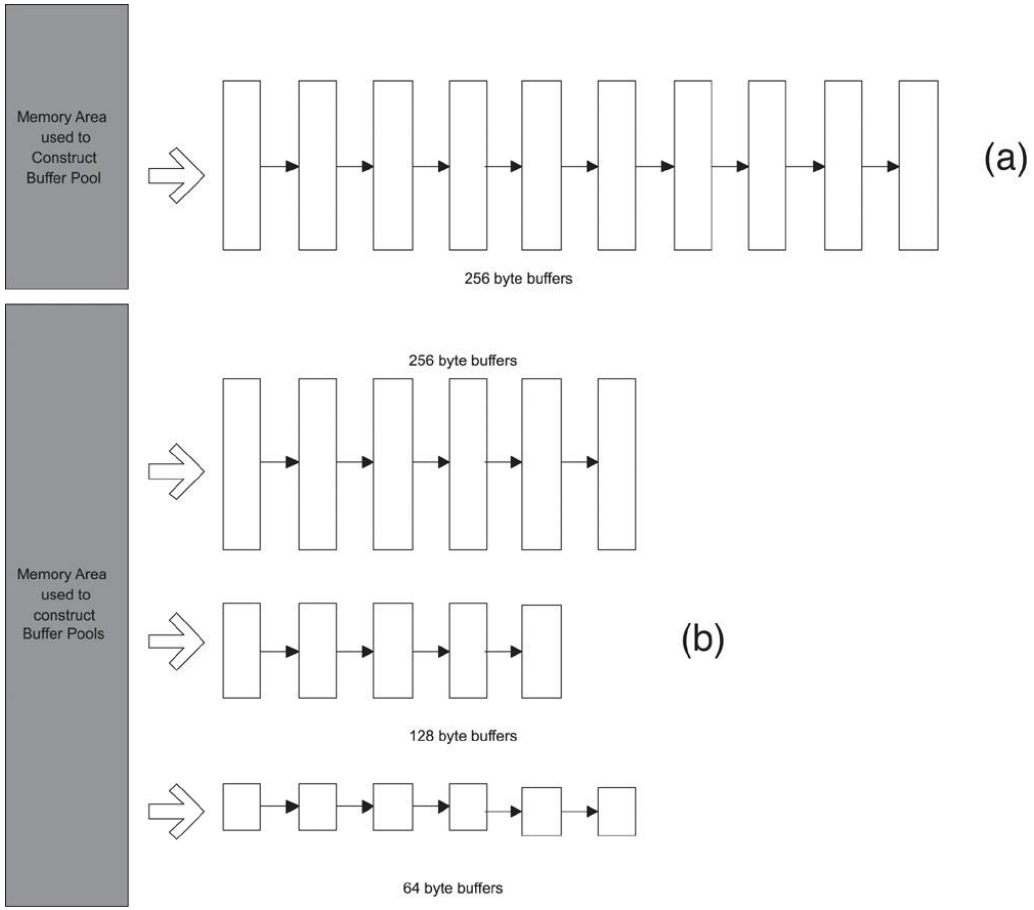
Buffer Management

- Global vs Local Buffer Management
  - Global uses a single buffer pool for the entire system with size determined apriori
    - ADV: Easy to manage memory DISADV: Lack of isolation between modules
  - Local is indicative of each module operating from it's own buffer pool
    - ADV: Decentralized and flexible DISADV: No uniform view of buffer requirements
  - Global buffer management libraries offer the best of both worlds
    - Techniques and mechanisms are global but actual memory allocation is from local buffer pools

Buffer Management

- Single vs Multiple buffer pools
  - In single scheme all buffers in the pool are of same size. In multiple scheme, buckets of varying sizes are created where all buffers in one bucket are of the same size
  - Multiple scheme allows designer flexibility in picking right size buffer and hence reduces memory wastage
- How to determine the buffer size?
  - Most common frame data size to be used as buffer size
  - When frames exceed buffer size, buffer chain can be created to store all data (mbuf in UNIX)
  - No matter what the buffer size, not possible to eliminate internal fragmentation

Buffer Management



Buffer Management

Example:

- Received frame size is 300 bytes
- buffer size 64 bytes:
  - Number of buffers required = 300/64 = 4 + 1 = 5 buffers
  - Size of data in the last buffer = Modulo 300/64 = 44 bytes
  - Unused data in the last buffer = 64 - 44 = 20 bytes

BSD Buffer Scheme

- Berkeley Systems Distribution (BSD) buffer library is called mbuf
- mbuf is a two level hierarchy scheme for buffer management
- mbuf routines designed for scatter/gather operations
  - Data from a frame is scattered in different memory locations and has to be gathered to construct full packet
- mbuf is 128 bytes long with 108 bytes for data
  - When data exceeds 108 bytes, mbuf points to an external data area called mbuf cluster.
  - Data is stored in the internal data area or external mbuf cluster but never in both areas.
  - mbuf and mbuf_cluster form the two levels of hierarchy.

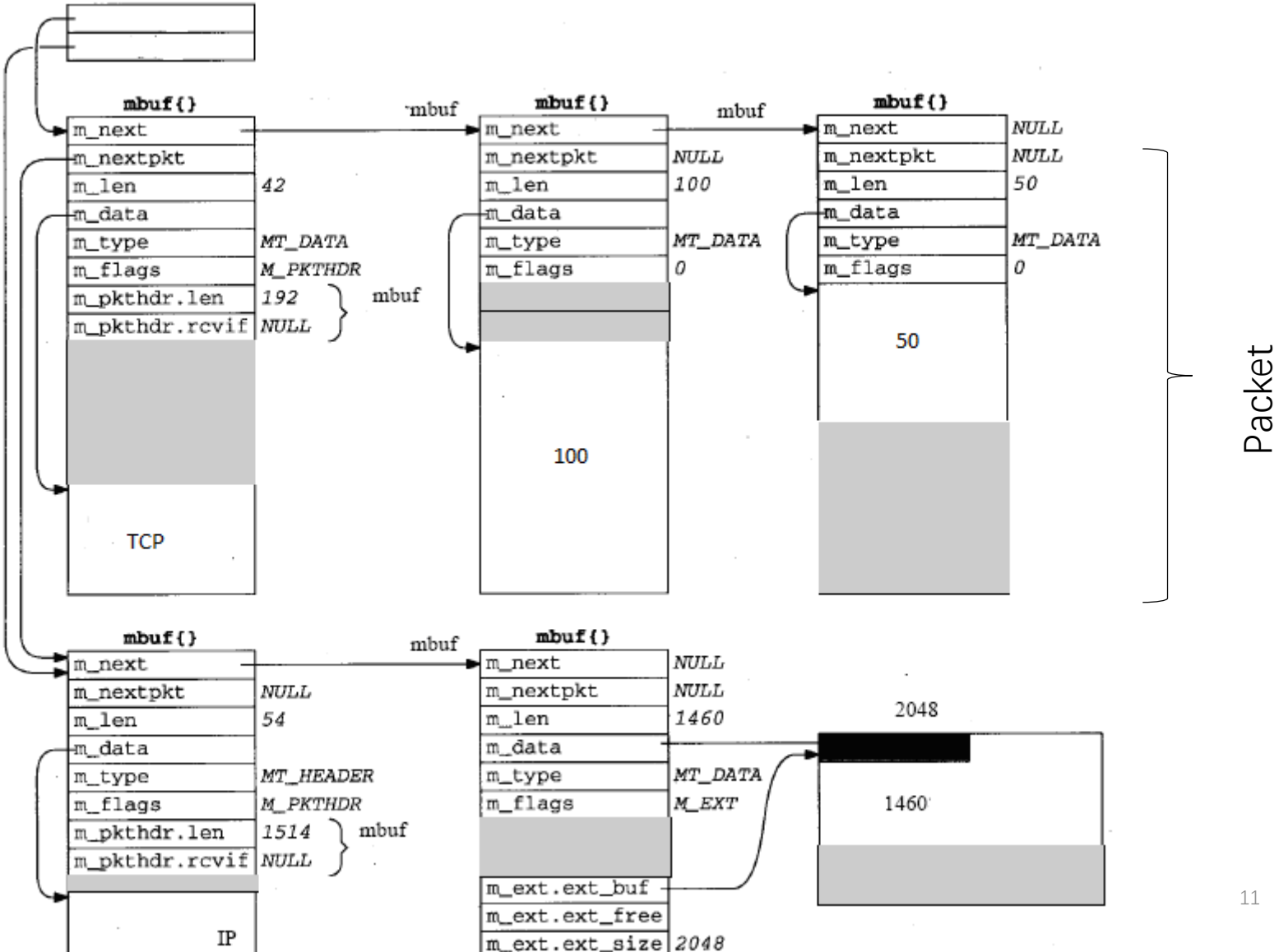
BSD Buffer Scheme

- mbuf uses m_data pointer to indicate valid data
  - which can be incremented to delete memory at the beginning of the buffer
  - Decrementing m_len deletes memory at end of the buffer
  - Manipulation of these pointers help eliminate the need for data copying (TCP->IP) or (IP->TCP)
- Multiple mbuf's can point to the same mbuf cluster
  - This helps to send same frame to different interfaces without the need to copy the data each time
- mbuf library includes routines for allocation, freeing, copying of data and deleting data at beginning and end of buffer

BSD Buffer Scheme

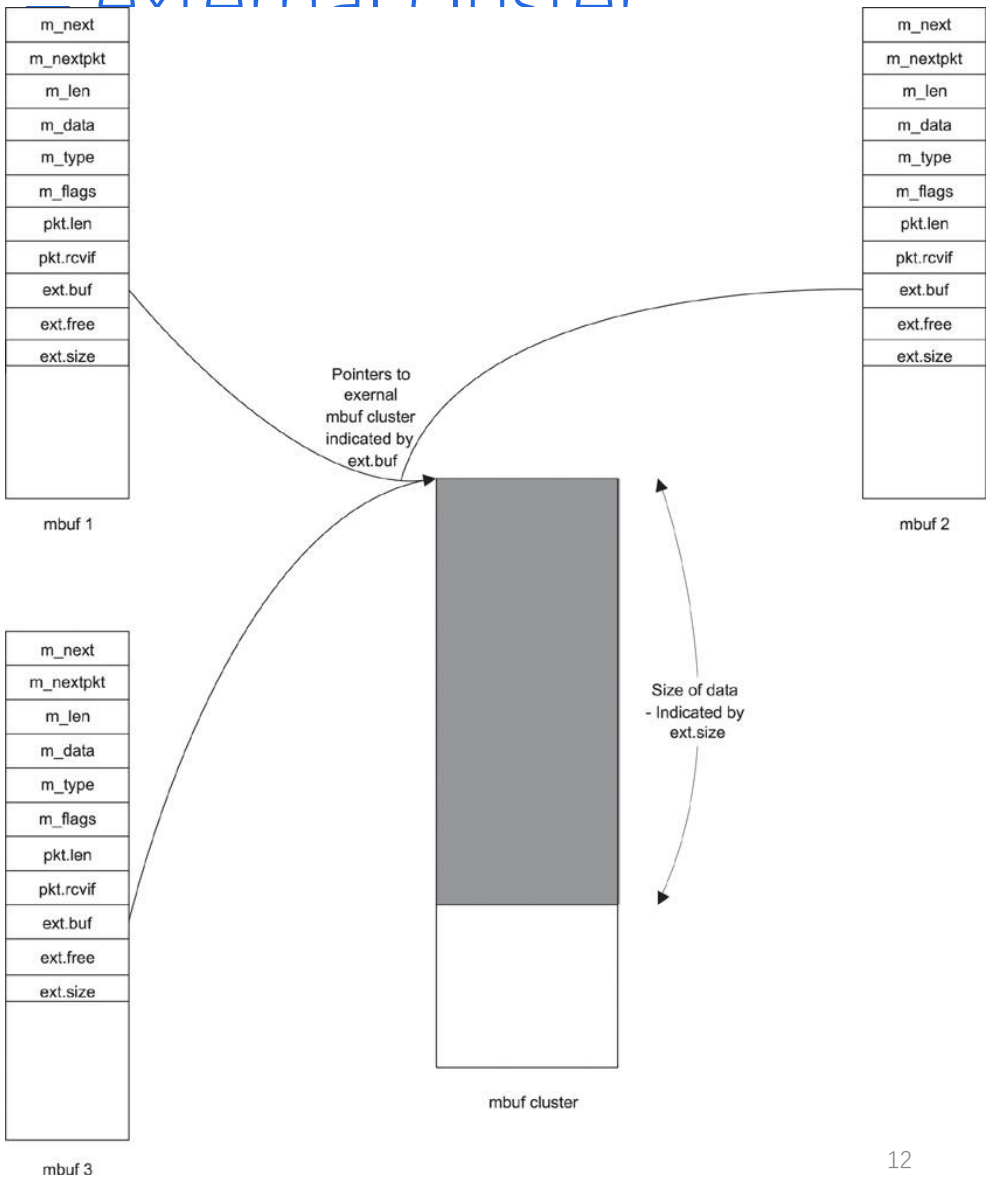
- An mbuf can be linked to another mbuf with the m_next pointer.
- Multiple mbufs linked together constitute a chain, which can be a single message like a TCP packet.
- Multiple TCP packets can be linked together in a queue using the m_nextpkt field in the mbuf.
- Each mbuf has a pointer, m_data, indicating the start of "valid" data in the buffer.
- The m_len field indicates the length of the valid data in the buffer.

Buffer Management



BSD Buffer Scheme – external cluster

Link multiple mbufs
to a single mbuf cluster



mbuf Library Routines

Function Name	Description and Use
m_get	To allocate an mbuf mptr = m_get (wait, type)
m_free	To free an mbuf m_free (mptr)
m_freem	To free an mbuf chain m_freem (mptr)
m_adj	To delete data from the front or end of the mbuf m_adj (mptr, count)
m_copydata	To copy data from an mbuf into a linear buffer m_copydata (mptr, startingOffset, count, bufptr)
m_copy	To make a copy of an mbuf mptr2 = m_copy (mptr1, startingOffset, count)
m_cat	To concatenate two mbuf chains m_cat (mptr1, mptr2)

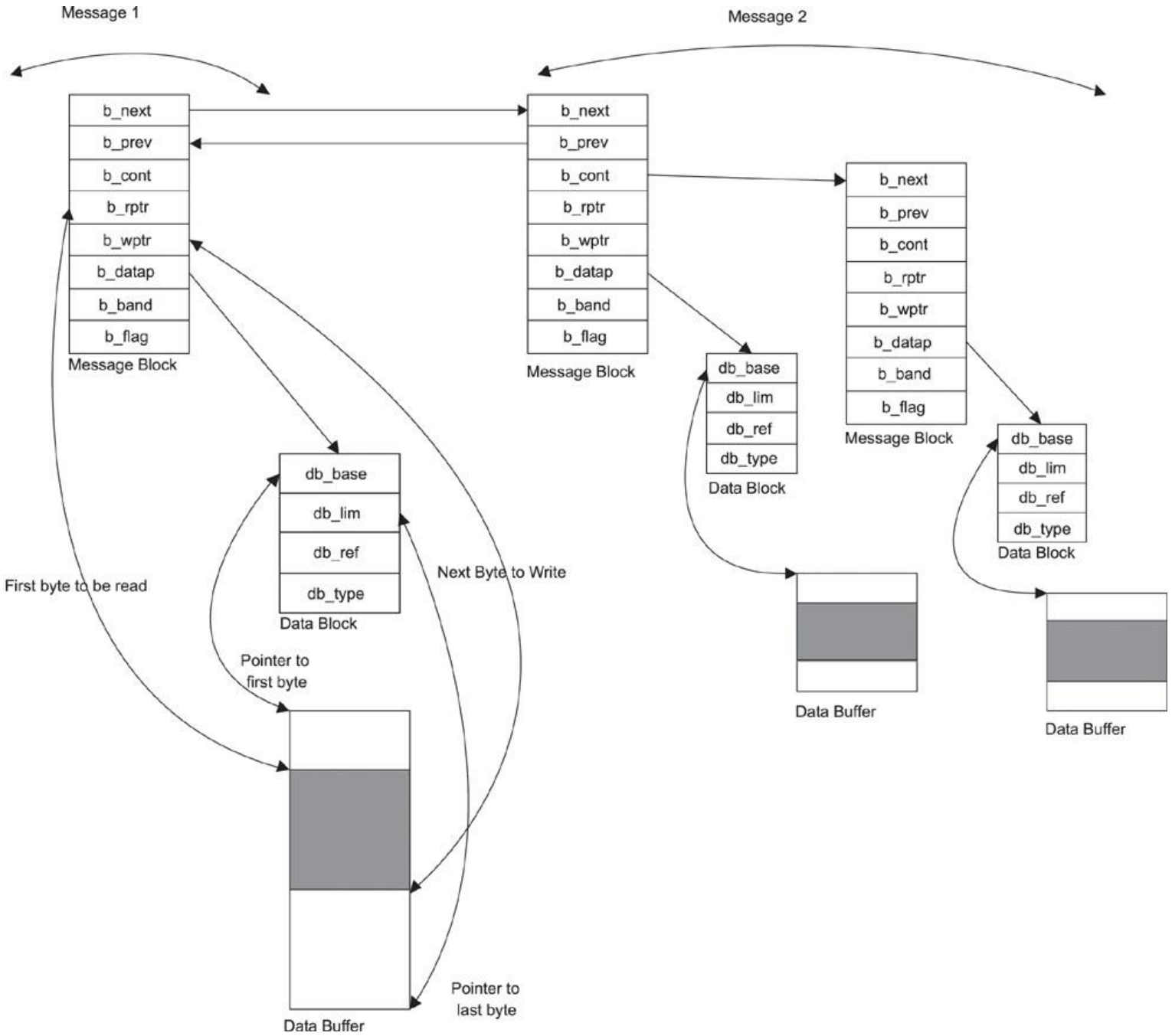
Buffer Management

- In 3-level hierarchy we have three types of structures
  - Message Block
  - Data Block
  - Data Buffer
- Message Block
  - Allocated from DRAM and contains pointers to data block that changes dynamically during execution
  - Difference in number of allocations and releases indicates number of available memory blocks
  - LowWaterMark variable used to indicate low availability of memory blocks and is configurable

STREAMS Buffering scheme

- STREAMS Buffering is a three level hierarchy scheme
- In data block db_base and db_lim point to first and last byte of data and can be used just like mbuf to delete data from beginning or end of buffer
- STREAMS uses linking to modify data without copy, concatenate or duplicate operations on the buffer

STREAMS Buffering scheme


Message 1
Message 2
b_next
b_prev
b_cont
b_rptr
b_wptr
b_datap
b_band
b_flag
Message Block
db_base
db_lim
db_ref
db_type
Data Block
First byte to be read
Next Byte to Write
Pointer to first byte
Pointer to last byte
Data Buffer

STREAMS Buffering scheme

- Comparison of 2-level and 3-level schemes
  - 2-level is simple and just one level of indirection gets data but 3-level needs one more level of indirection
  - 3-level needs more memory to store message blocks
  - In a three-level hierarchy, the message pointer (b_datap) does not need to change to add data at the beginning of the message. The message block now points to a new data block with the additional bytes.
    - This is transparent to the application since it continues to use the same pointer for the message block. With a two-level hierarchy, this could involve allocating a new mbuf at the head of the mbuf chain and ensuring that applications use the new pointer for the start of the message.

Buffer Management

- EXCEPTION conditions in Buffer Management
  - Buffer pool empty: Modules slow in processing, memory leaks due to bad modules, data rate not in sync with system speed
  - Modules may be slow because their priority is low, processing logic is complex or other reasons and this could lead to buffers being held up
  - Errant modules are those that are buggy and cause memory leaks and the low water mark variable will help identify these issues
  - System not keeping up with data rates can lead to congestion. There are a variety of schemes to handle congestion implementable in software and hardware.

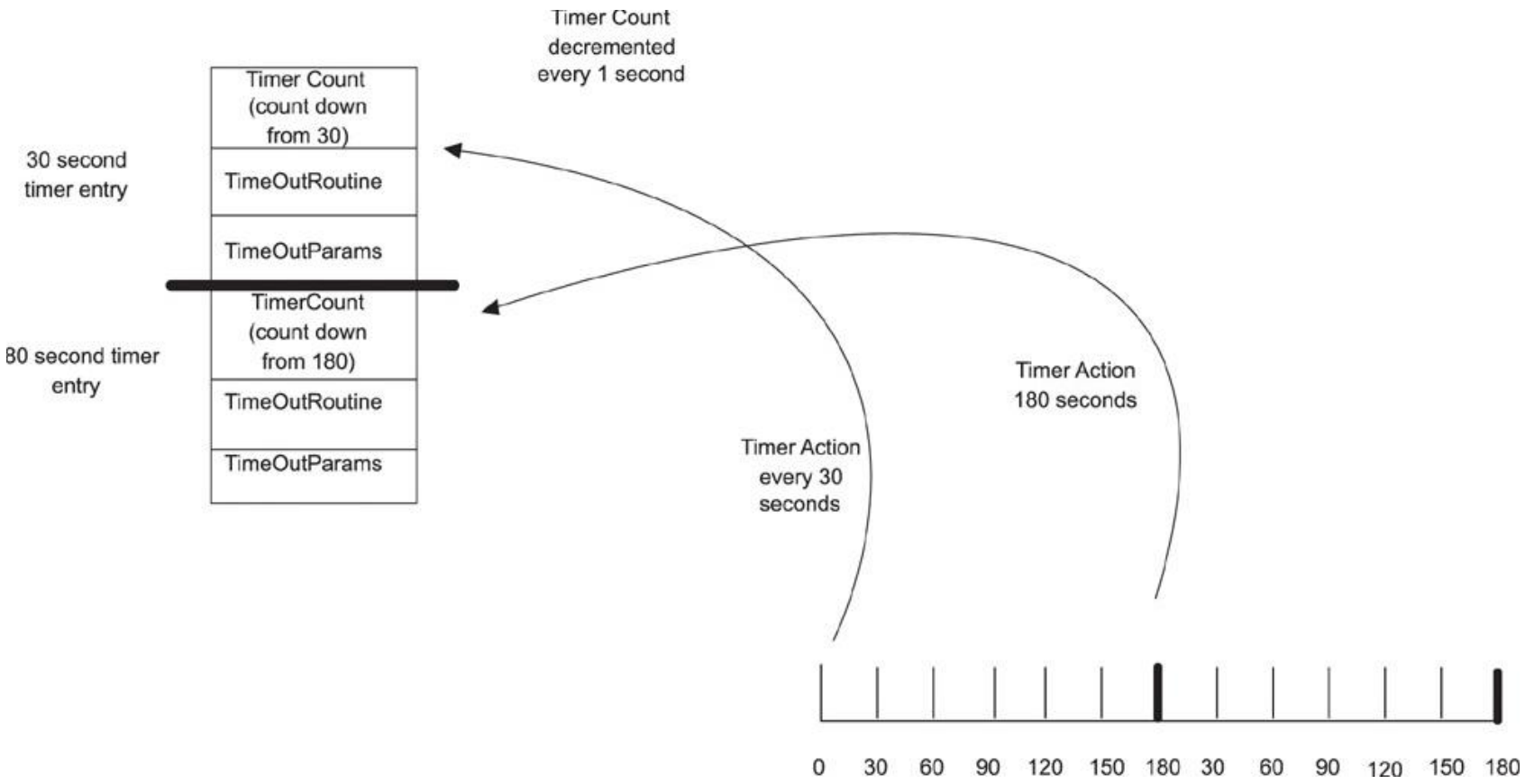
Timer Management

- Three significant uses for timers
  - For periodic execution of certain tasks
  - Timeout of tasks waiting for messages
  - For tasks that need to be fired at a particular time (one-shot timers)

Timer Management

- Timer management at task level
  - An RTOS typically offers facilities to manipulate timers based on system clock
  - Timeout routine is invoked when a timer finishes a count of N clock ticks
  - Timers require dynamic requirements and both table based implementation and linked list implementation have their drawbacks due to continual decrement operation.
  - Two types of timers:
    - Continuous timer: The value is reset and countdown to N is started again and the timeout routine invoked again.
    - One-shot timers: After the countdown has finished and timeout routine is called, the timer is removed from the tables

Timer Management



Ticker object

- The Ticker interface is used to setup a recurring interrupt to repeatedly call a designated function at a specified rate.

- Avoids the requirement for the main code to continuously analyze the timer to determine whether it was the right time to execute a specified function.

Timer Management – A simple implemenation

```
AppClock( ) { ……..
    static unsigned long  tickCount = 0;
    tickCount++;
    If (tickCount == N1)
    {
       tickCount = 0;
       Notify (myTask);
    }
}
```

```
myTask{
   N1=100;
   setTimer(N1);
   While(1)
   {
        Wait_for_event()
        if (event==TIMER)
           processTimerExpired();
    }
}
```

- AppClock () is a routine which is called from the timer interrupt on each OS TICK
- If OS TICK=10 ms, Task1 will be notified every 1s
- Notify(myTask) sends a timer event to the myTask that schedules the timer.
- Can get complex and cumbersome to manage if we have many tasks and many timers.

Better alternative

```
AppClock (  {
    tickCount++;
    TimerTaskNotify (); // send timer event message to timer task
}
```

- AppClock () is a routine which is called from the timer interrupt on each OS TICK
- A single timer management task called TimerTask() will manage all timers created by all tasks.

Challenges with timer interrupts

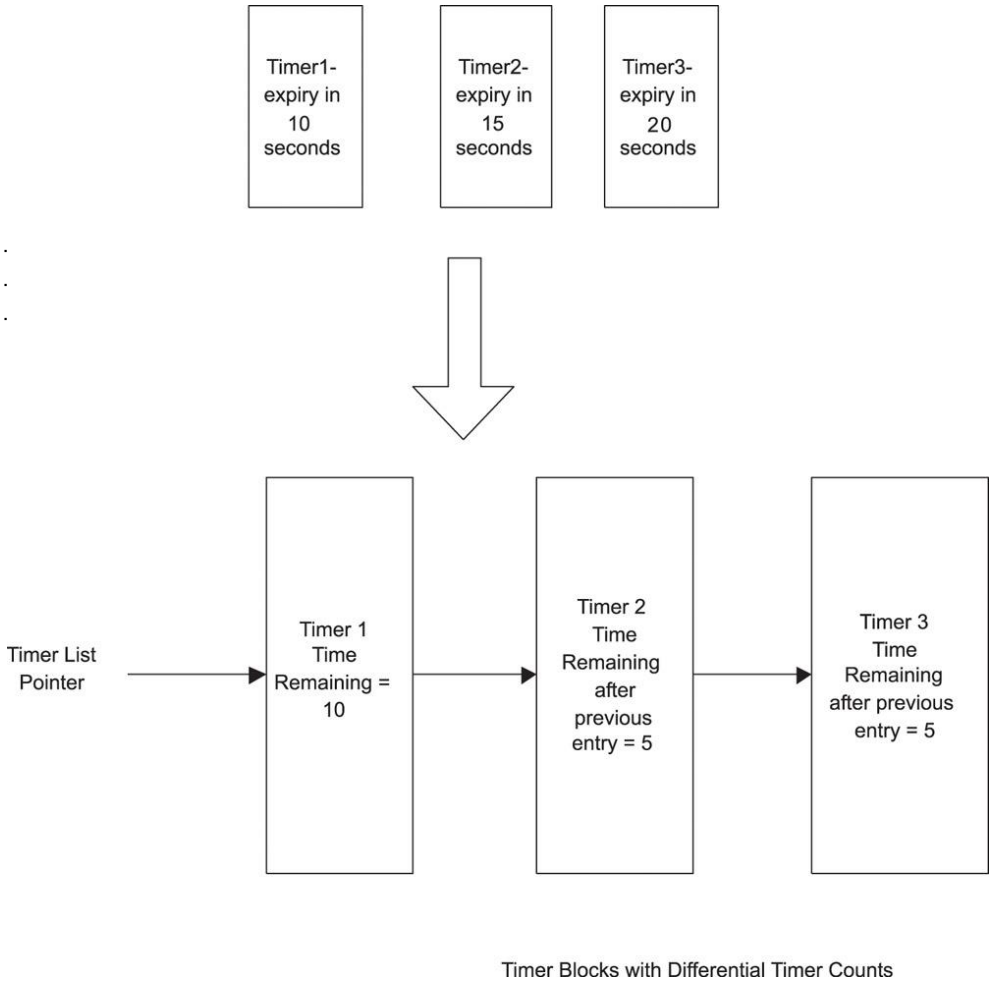
- Be careful with the amount of code and how long it takes to execute.
  - If we need to run a task every 1 ms, that task must take less than 1 ms second to execute, otherwise the timing would overrun and the system would go out of sync.
  - We sometimes need to prioritize the tasks: e.g. does a 1ms task run before a 100ms task? (because after 100ms, both will want to run at the same time).
  - This also means that pause, wait or delays (i.e. timing control by 'polling') cannot be used within scheduled program designs.

Differential timers

- Differential timers
  - In a differential timer scheme, the lowest timer count serves as the base and all other timers are stored with the incremental difference
  - The advantage here is the timer count is decremented only for the base timer and it automatically applies to all timers

Differential timers

startTimer(10, task1Timer, param …
startTimer(15, task1Timer, param …
startTimer(20, task1Timer, param …



Timer Blocks with Differential Timer Counts

Homework

- Is there any drawback with using differential timers?
- Can you come up with a rule for updating the time remaining for other timers already in the list when a timer is inserted in the list of differential timers?
- Would task priority mater when you insert differential timers in the list?
- Can you show an example to demonstrate your answers above?

Timer Management

- This is done in task's main loop every time task is scheduled

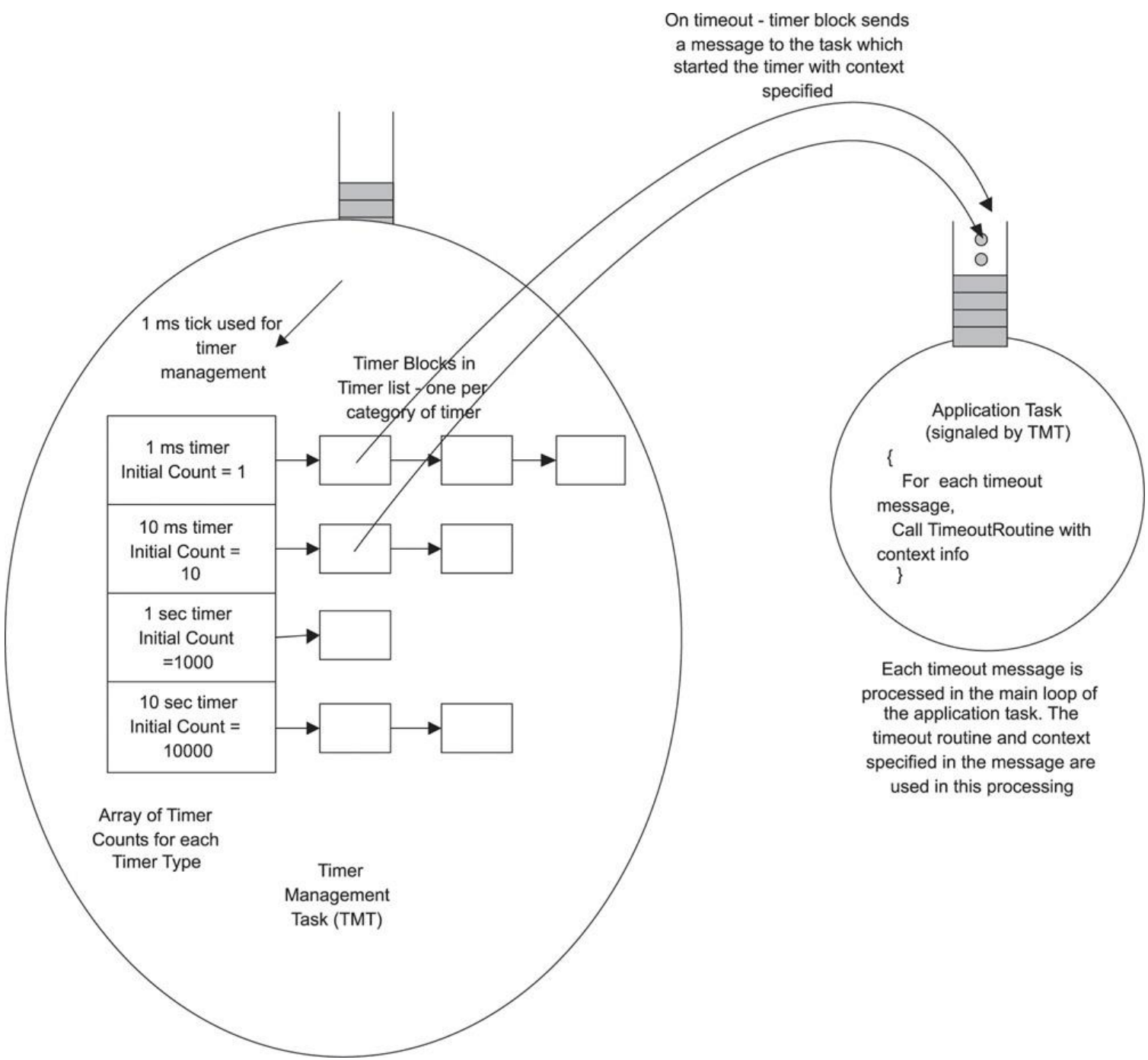
```
While(1)
{
     Wait_for_event()
     if (event==timer_expired)
          processTimers();
}

ProcessTimers ()
{
Decrement the current timer count in the first entry of the timer list;
If the count is 0
{
    For all the entries in the timer list whose current timer count is zero
    {
        Process timer expiry by calling the timeout
           routine with context provided in the timer block;
        Alternatively, it can send a timer signal event to
        the task it created the timer with context provided in timer block
     }
}
```

Timer Management Task

- Timers may be created and used individually for each task
  - This is flexible but unmanageable with more tasks
- Single timer management task (TMT) is scheduled after each timer tick interrupt
  - Timer tick interrupt is the shortest timer tick possible across all tasks as a base (e.g. 1 OSTICK).
  - It may maintain counters for simulating N* OSTICK timers. For each of these tick types, there are separate timer lists.
  - Tasks request a timer from the TMT which invokes the appropriate notification function when timer expires

Timer Management Task

- Assume the need for a 1-millisecond tick, a 10-millisecond tick, and a 1-second tick.
- The TMT will be notified of a 1-millisecond tick only.
- It will maintain counters for simulating a 10-millisecond and 1-second timer.
- For each of these tick types, there are separate timer lists.


On timeout - timer block sends a message to the task which started the timer with context specified
1 ms tick used for timer management
Timer Blocks in Timer list - one per category of timer
1 ms timer Initial Count = 1
10 ms timer Initial Count = 10
1 sec timer Initial Count =1000
10 sec timer Initial Count = 10000
Array of Timer Counts for each Timer Type
Timer Management Task (TMT)
Application Task (signaled by TMT)
{
For each timeout message,
Call TimeoutRoutine with context info
}
Each timeout message is processed in the main loop of the application task. The timeout routine and context specified in the message are used in this processing

Timer Management Task

- Individual timer management is flexible but each timer needs it's own timer processing logic- Timer management is Task's responsibility - Implementation of Tick counter management is OS's responsibility call from Timer ISR
- TMT centralizes timer tick management but needs a messaging scheme to interact with the tasks on timer expiration - Suitable if the granularity of a tick varies.

System Issues

- Each timer management scheme can allocate and free large number of timer blocks that need space to store application context
  - Connection oriented protocols like TCP require a large number of timers and hence need significant amount of memory
- Checklist for timer management strategy
  - Use RTOS timer ISR for notification of the smallest timer tick
  - Choose minimal number of application timer ticks
  - Implement a differential time scheme