Real-Time & Embedded Systems

Agenda

- uC questions / wrapup
- RTOS (Chapter 3)

Operating systems taxonomy

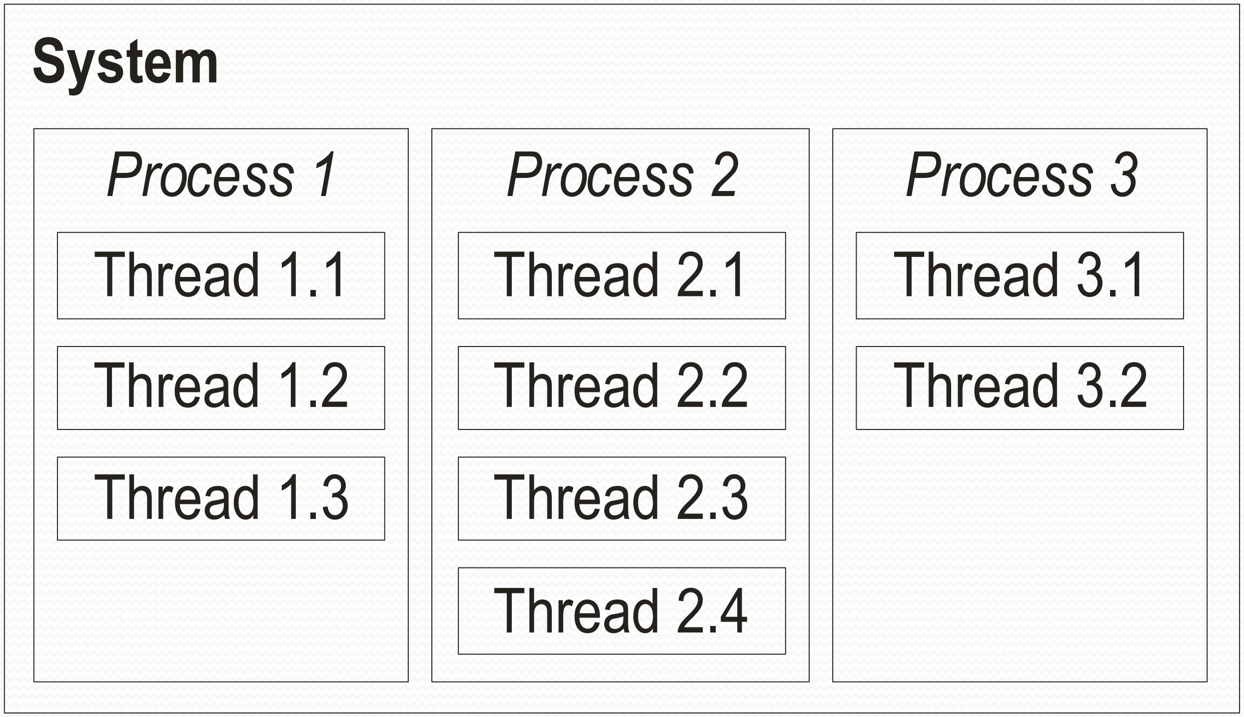
Users

Operating System	User Interface, Security Features, and File Management
Executive	Privatized Memory Blocks, Input/Output Services, and Supporting Features
Kernel	Intertask Communication and Synchronization
Microkernel	Task Scheduling and Dispatching

Hardware

The role of the kernel in operating systems. Moving up the taxonomy from the low-level micro-kernel to the full-featured operating system shows the additional functionality provided and also indicates the relative closeness to hardware versus human users.

Processes, and multiple threads



Polled loop systems

- Simplest form of Real-time “kernel”.
- Used for fast reaction to single events
- Do not require interrupts.
- Simple while loop used to wait for an event signaled via DMA from a hardware device.
- Handler should clear event before servicing to catch bursts.
- These systems are easy to construct and analyze and thus to guarantee response times.

Polled loop systems

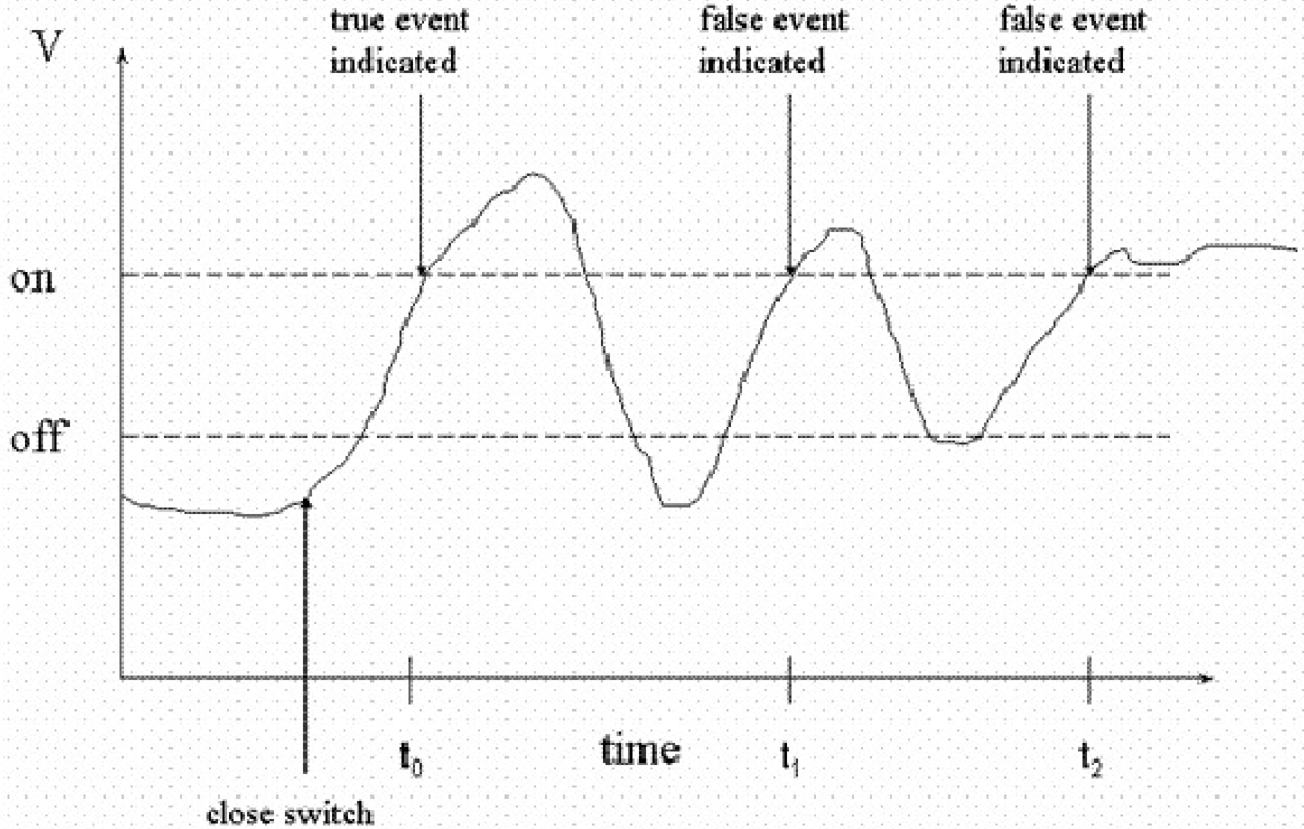
Hardware causes event via memory mapped I/O, DMA or a pin input

```
loop     {                                 /* do forever     */
if (packet_here)                           /* check flag     */
         {
                  packet_here=0;           /* reset flag     */
                  process_data();          /* process data */
         }
}
```

Synchronized polled loops

- Polled loop response may be so fast that false events are induced by premature resetting of the flag.
- Switch bounce phenomenon in physical systems is a cause of such problems.
- Delay can be a no-op, loop, or interrupt (clock) driven delay.
- Increases response times-- but it's worth it.

Synchronized polled loops



Switch bounce phenomenon. The switch is closed at time t0, signaling the event, however, due to the ringing of the signal and the edge triggered logic several false events could be indicated at times t1, and t2.

Synchronized polled loops

wait for switch to settle

```
loop      {                          /* do forever    */
          if(flag)                   /* check flag    */
          {
              pause(20);             /* wait 20 ms    */
              flag=0;                /* reset flag    */
              process_event();       /* process event */
          }
}
```

Cyclic executives

- Simplest form of “multitasking.”
- Processes are run in round-robin fashion.
- Processes can be subdivided into “minor cycles” using FSM driven code.
- A small number of short processes will appear to be running concurrently.
- Easy to analyze for response times.
- Some people call periodic interrupt driven systems cyclic executives.

Cyclic executives

```
loop    {          /* do forever   */
          check_for_keypressed();
          move_aliens();
          check_for_keypressed();
          check_for_collison()
          check_for_keypressed();
          update_screen();
        }
}
```

Cyclic executive for “Space Invaders.”

Cyclic executives

- Not very flexible.
- Response times are very slow.
- All tasks run at the same “rate.”
- Tasks can be broken into states or phases to reduce response times an to create “minor” cycle.
- Can use FSM driver presented previously.
- Still does not require interrupts.

Coroutines

- Simplest form of “fairness scheduling.”
- Processes voluntarily exit at strategic points of execution
- Used with FSM.
- Inter-process communication through global variables.
- Sometimes called cooperative multitasking.
- Example: early versions of CICS.

Coroutines

```
void process_a(void)

{

loop {
      switch(state_a)
       {
             case 1:      phase_a1();
                          break;
             case 2:      phase_a2();
                          break;
             case 3:      phase_a3();
                          break;
             case 4:      phase_a4();
                          break;
             case 5:      phase_a5();
                          break;
      }
    }

}
```

```
void process_b(void)

{

loop {
      switch(state_b)
      {
             case 1:      phase_b1();
                          break;
             case 2:      phase_b2();
                          break;
             case 3:      phase_b3();
                          break;
             case 4:      phase_b4();
                          break;
             case 5:      phase_b5();
                          break;
      }
    }
}
```

Interrupt service routines

- Hardware interrupt: a signal generated by a peripheral device and sent to the CPU. The trigger is an electrical signal from an external device.
- Software interrupt: similar to the hardware interrupt, and it causes one code module to pass control to another. Trigger of is the execution of a machine language instruction.
- An exception is a software interrupt that is internal to the CPU and triggered by a software program's attempt to perform an unexpected or illegal operation.
- All three interrupts cause the CPU to transfer execution to a known location and then execute an interrupt service routine (ISR).

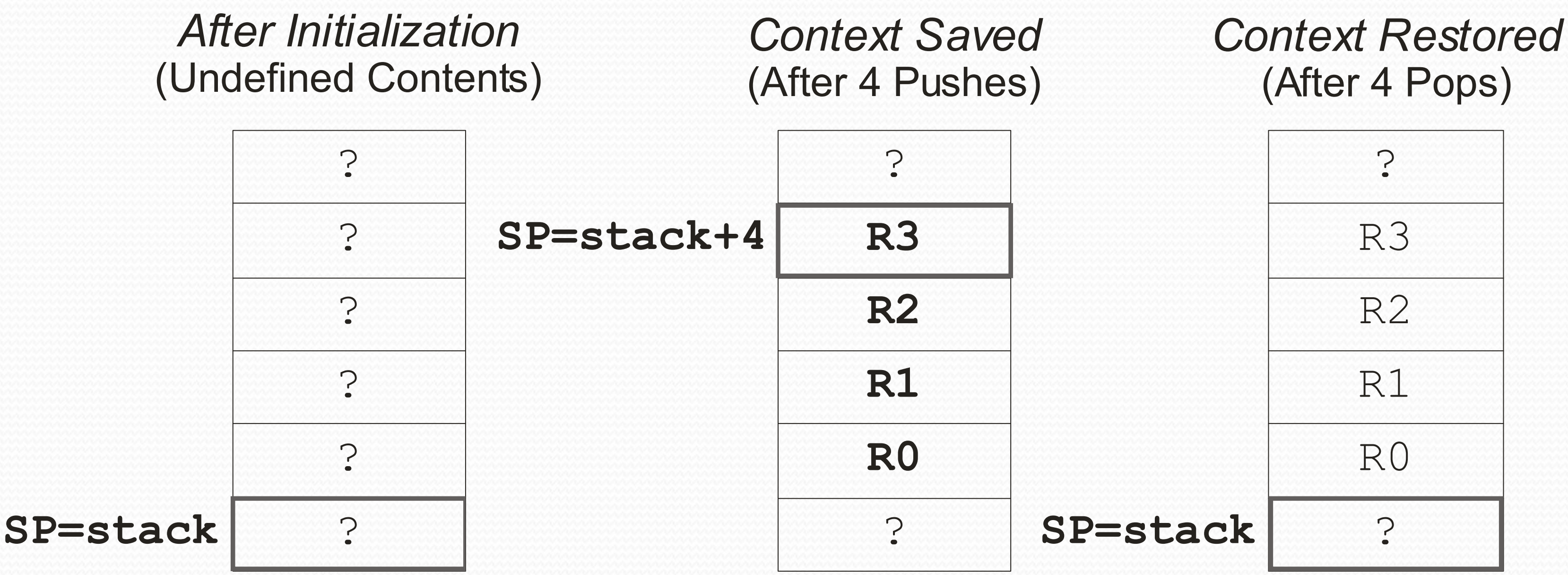
Interrupt service routines

- Hardware interrupts are asynchronous in nature.
- Access to resources shared with an ISR is usually controlled by disabling interrupts in the application around any code that reads or writes to the resource.
- Synchronization mechanisms cannot be used in an ISR because the ISR cannot wait indefinitely for a resource to be available.
- If the ISR takes too long to process an interrupt, the external device may be kept waiting too long before its next interrupt is serviced.
- In all ISRs a snapshot of the machine—the context—must be preserved upon switching tasks so that it can be restored upon resuming the interrupted process.

Context switching

- Context switching is the process of saving and restoring sufficient information for a real-time task so that it can be resumed after being interrupted.
- The context is ordinarily saved to a stack data structure.
- Context switching time is a major contributor to response time and therefore must be minimized.
- The rule for saving context is simple: save the minimum amount of information necessary to safely restore any process after it has been interrupted.

Context switching



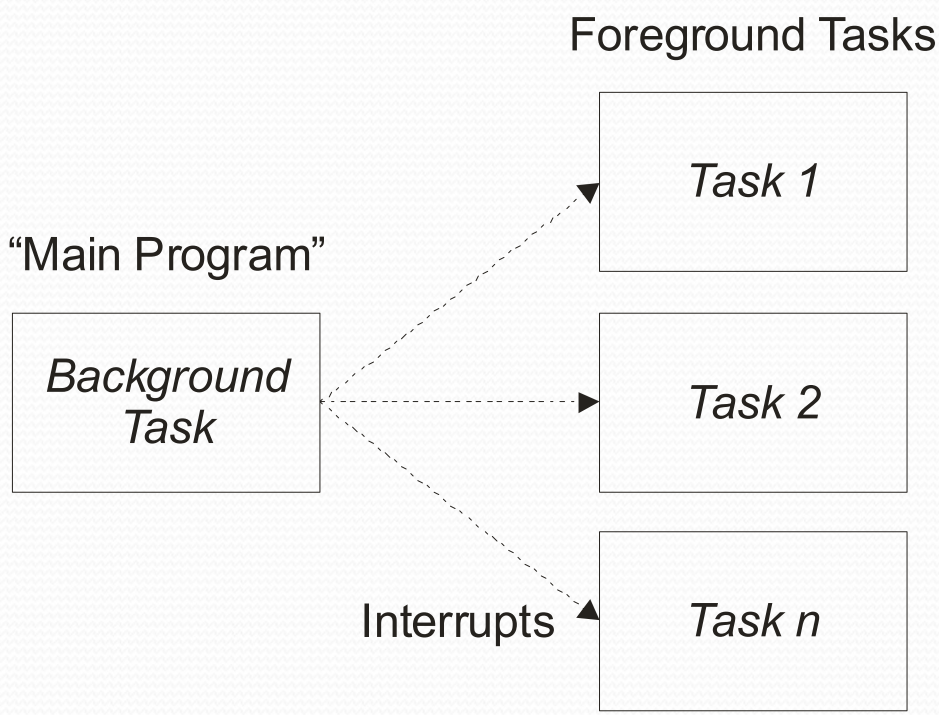
Context switching

- Context usually includes:
  - contents of general registers
  - contents of the program counter
  - contents of coprocessor registers (if present)
  - memory page register
  - images of memory-mapped I/O locations (mirror images)
- Interrupts are disabled during context-switching. Sometimes a partial context switch is used to handle a burst of interrupts, to detect spurious interrupts, or to handle a time-overloaded condition.
- The stack model for context switching is used mostly in embedded systems where the number of real-time or interrupt-driven tasks is fixed.
- In the stack model, each interrupt handler is associated with a hardware interrupt and is invoked by the CPU, which vectors to the instruction stored at the appropriate interrupt-handler location.
- The context is then saved to a specially designated memory area that can be static, in the case of a single-interrupt system, or a stack, in the case of a multiple-interrupt system.

Preemptive priority systems

- Preemptive priority systems use preemption (prioritized interrupts). The priorities assigned to each interrupt are based on the urgency of the task associated with the interrupt.
- Prioritized interrupts can be either fixed priority or dynamic priority.
  - Fixed-priority systems are less flexible since the task priorities cannot be changed.
  - Dynamic-priority systems can allow the priority of tasks to be adjusted at run-time to meet changing process demands.
- Preemptive priority schemes can suffer from resource hogging by higher-priority tasks leading to a lack of available resources for lower-priority tasks. This is called starvation.
- Rate-monotonic systems are those fixed priority periodic real-time systems where the priorities are assigned so that the higher the execution frequency, the higher the priority.

Foreground/background systems



Foreground/background systems are the most common architecture for embedded applications. They involve a set of interrupt-driven or real-time processes called the foreground and a collection of non-interrupt driven processes called the background.

Background processing

- The background consist of all non-interrupt driven tasks. Includes anything that is not time critical.
- Foreground tasks are interrupt driven tasks and are time critical.
- Typical background tasks include:
  - low-priority self-testing
  - display updates
  - logging to printers, non-volatile memory
  - interfaces to slow devices

Initialization

- Initialization of the foreground/background system:
  - disable interrupts
  - set up interrupt vectors and stacks
  - perform self-test
  - perform system initialization
  - enable interrupts
- Initialization is actually the first part of the background process.
- It is important to disable interrupts because many systems come up with interrupts enabled while time is still needed to set things up.
- Initialization includes initializing the appropriate interrupt vector addresses, setting up stacks, and initializing any data, counters, arrays.
- Perform self-diagnostic tests before enabling any interrupts.
- Only then can real-time processing can begin.

Full-featured real-time operating systems

- The foreground/background solution can be extended into an operating system by adding additional functions such as network interfaces, complicated device drivers, and complex debugging tools.
- These types of systems are readily available as commercial products.
- Such systems rely on a complex operating system using round robin, preemptive priority, or a combination of both schemes to provide scheduling; the operating system represents the highest priority task, kernel, or supervisor.
- Commercial real-time operating systems are most often of this type. The task-control block model is most often used in these types of systems because the number of real-time tasks is indeterminate and dynamic.

The task control block model

- **Each task is associated with a data structure – a task control block (TCB).**
- **TCB contains context. The system stores TCBs in one or more data structures, such as a linked list.**
- **TCB model can be used in round-robin, preemptive priority or combination systems, although it is generally associated with round-robin systems with a single clock.**
- **Is the most popular method for implementing commercial, full-featured, real-time operating systems.**
- **Also used in interactive on-line systems where tasks (associated with users) come and go.**

The task control block model

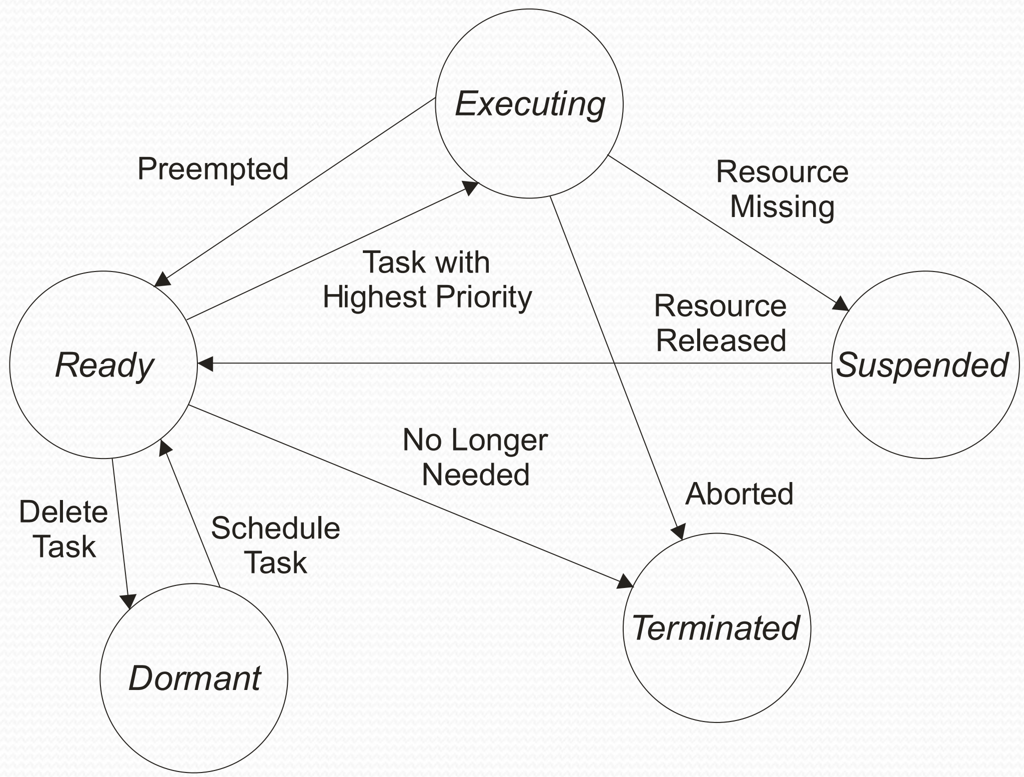
- **The operating system manages TCBs by keeping track of the status of each task.**
- **A task can typically be in any one of the following states:**
  - **executing**
  - **ready**
  - **suspended (or blocked)**
  - **dormant (or sleeping)**

Typical task control block

TCB

Task Identifier
Priority
Status
Work Registers
Program Counter
Status Register(s)
Stack Pointer
Pointer to Next TCB

The task control block model



A process state diagram as a partially defined finite state machine.

The task control block model

- Every hardware interrupt and every system level call (such as a request on a resource) invokes the real-time operating system.
- The operating system is responsible for maintaining a linked list containing the TCBs of all the ready tasks, and a second linked list of those in the suspended state.
- It also keeps a table of resources and a table of resource requests.

The task control block model

- In the TCB model tasks track their own resources.
- The TCB model is very flexible.
- The main drawback of the task-control block model is that when a large number of tasks are created, the overhead of the scheduler can become significant.

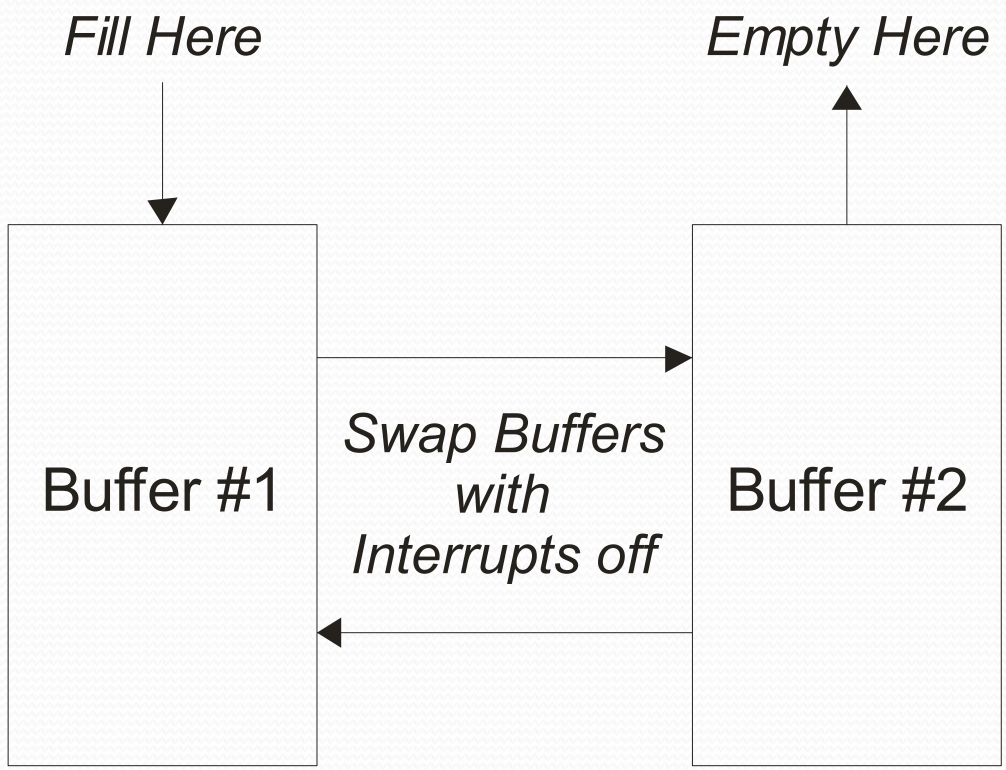
Buffering data

- Mechanisms are needed to pass data between tasks in a multitasking system when production and consumption rates are unequal.
- Global variables are simple and fast, but have collision potential.
- Example, one task may produce data at a constant 100 units per second, whereas another may consume these data at a rate less than 100 units per second.
  - Assuming that the production interval is finite (and relatively short), the slower consumption rate can be accommodated if the producer fills a storage buffer with the data.
  - The buffer holds the excess data until the consumer task can catch up.
  - The buffer can be a queue or other data structure, including an unorganized mass of variables.
  - If consumer task consumes this information faster than it can be produced, or if the consumer cannot keep up with the producer, problems occur.
- Selection of the appropriate size buffer and synchronization mechanisms is critical in reducing or eliminating these problems.

Time relative buffering

- Can use global variables for double buffering or ping-pong buffering.
- Used when time-relative (correlated) data need to be transferred between cycles of different rates, or when a full set of data is needed by one process but can only be supplied slowly by another process.
- Variant of the classic bounded buffer problem in which a block of memory is used as a repository for data produced by "writers" and consumed by "readers."
- Further generalization is the readers and writers problem in which there are multiple readers and multiple writers of a shared resource

Time relative buffering

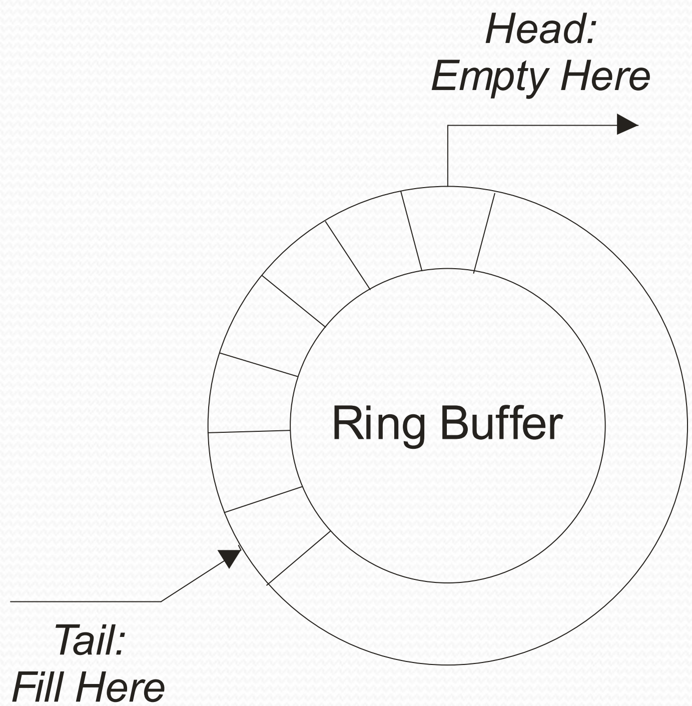


Double buffering configuration. Two identical buffers are filled and emptied by alternating tasks. Switching is accomplished either by a software pointer or hardware discrete.

Ring buffers

- A circular queue or ring buffer can be used to solve the problem of synchronizing multiple reader and writer tasks.
- Ring buffers are easier to manage than double buffers or queues when there are more than two readers or writers.
- Simultaneous input and output to the list are achieved by keeping head and tail indices.
- Data are loaded at the tail and read from the head.
- Can be used in conjunction with a counting or binary semaphore to control multiple requests for a single resource such as memory blocks, modems, and printers.

Ring buffers



A ring buffer. Processes write to the buffer at the head index and read data from the tail index. Data access is synchronized with a counting semaphore set to size of ring buffer.

Mailboxes

- A mutually agreed upon memory location that one or more tasks can use to pass data, or more generally for synchronization.
- Tasks rely on the kernel to allow them to write to the location via a `post` operation or to read from it via a `pend` operation.
- The difference between the pend operation and polling is that the pending task is suspended while waiting for data to appear -- eliminates the busy waiting condition.
- Mailboxes are available in most commercial RTOS.

Mailboxes

- The datum that is passed can be a flag used to protect a critical resource (called a key), a single piece of data, or a pointer to a data structure.
- In most implementations, when the key is taken from the mailbox, the mailbox is emptied.
- Since the key represents access to a critical resource, simultaneous access is precluded.
- Mailboxes are best implemented in systems based on the task control block model with a supervisor task.
- A table containing a list of tasks and needed resources is kept along with a second table containing a list of resources and their states.

Mailboxes

Task id #	Resource	Status
100	printer	has it
102	mailbox 1	has it
104	mailbox 1	pending

Task resource request table.

Resource	Status	Owner
printer 1	busy	100
mailbox 1	busy	102
mailbox 2	empty	none

Resource table used in conjunction with task resource request table.

Mailboxes

- When the supervisor is invoked by a system call or hardware interrupt, it checks the tables to see if some task is pending on a mailbox.
- If the key is available (key status is “full”), then than task must be restarted.
- If a task posts to a mailbox, then the operating system must ensure that the key is placed in the mailbox and its status updated to “full”.
- Other operations on the mailbox include the `accept` operation, which allows tasks to read the key if it is available, or immediately return an error code if the key is not available.
- In other implementations, the `pend` operation is equipped with a timeout, to prevent deadlocks.

Queues

- Some operating systems support `qpost`, `qpend`, and `qaccept` operations.
- In this case, the queue can be regarded as any array of mailboxes, and its implementation is facilitated through the same resource tables already discussed.
- Queues should not be used to pass arrays of data; pointers should be used instead.
- Queues are useful in implementing device servers where a pool of devices are involved:
  - The ring buffer holds requests for a device, and queues can be used at both the head and the tail to control access to the ring buffer.
  - This scheme is useful in the construction of device-controlling software.

Real-Time Systems Design and Analysis: Tools for the Practitioner

P. A. Laplante & S. J. Ovaska

Chapter 3

Part II