Tasks

Task Implementation and management

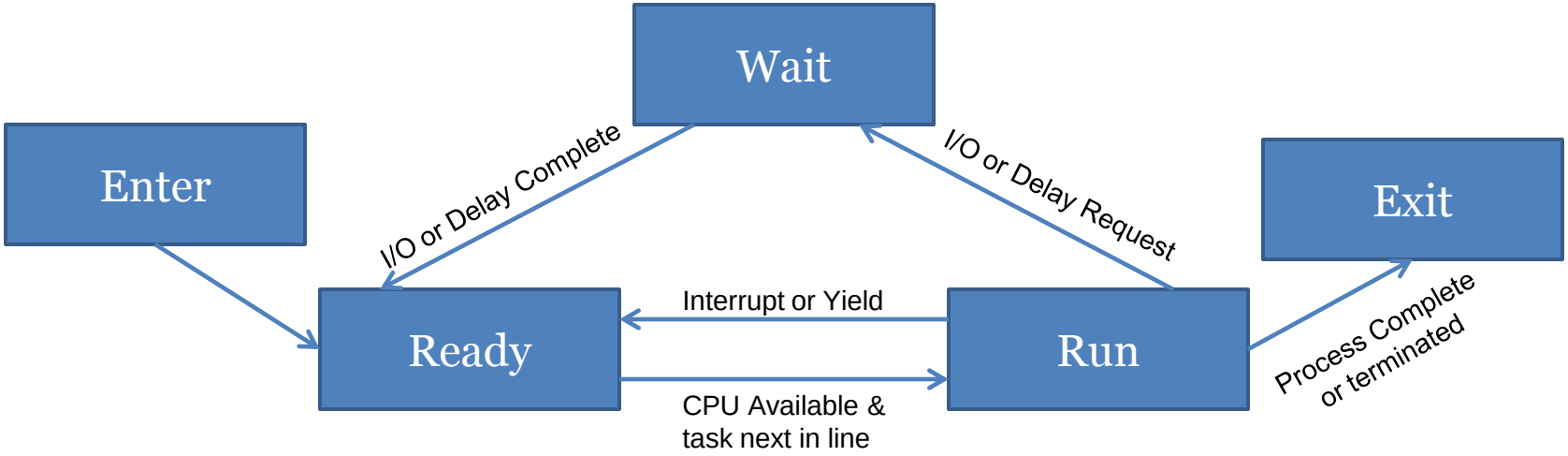
Tasks Vocab

- Absolute time - real world time
- Relative time - time referenced to some event
- Interval - any slice of time characterized by start & end times
- Duration - distance in time between start and end times
- Reactive systems - tasks triggered by events
- Time-based systems - tasks triggered by evolution of time
  - Absolute - triggered at specific time
  - Relative - task triggered before or after some reference or interval
- Periodic - tasks with constant duration between initiation
- Aperiodic - not periodic
- Execution time - CPU time to complete a task
- Jitter - variance of durations in intended periodic system
- Delay - amount of time between evoking event and start of task
- Hard deadline - if action doesn't occur by some time system is considered to have failed
- Hard real-time system - has at least one task with hard-deadline, focus of design on hard-deadlines

Tasks Vocab (2)

- Soft real-time - usually system must meet deadline "on average" e.g. to achieve an average throughout
- Firm real-time - mix, of hard and soft deadlines
- Predictability - how well we know the timing of tasks completing and starting
- Interarrival time - time between evoking events, esp. for aperiodic systems. Typically need to know bounds and/or average intervals between events to know if system can meet demands
- Priority - allows some task to be designated to run before others
- Schedulable - in real-time context, refers to a task that can be scheduled (added to the queue) and will meet its timing constraints
- Deterministically schedulable - in a system, means that it has been determined (can be guaranteed in-advance) that a given task will always be able to be scheduled and meet have its timing constraints met
- CPU Utilization – percentage of the time the CPU is being used as opposed to being idle (typically 40%-90%)

Scheduling Decisions



- Most common state transition conditions
- If run->ready is forced by OS interruption the system is preemptive, otherwise it is self-yielding or non-preemptive
- Non-preemptive OS tasks stop executing when they give up the CPU by completing or requiring a delay (wait state) or putting themselves into ready state (yield)

Scheduling Priority

- Non-preemptive and preemptive
  - Priority determines which task is selected when CPU becomes available
- Preemptive
  - Priority also determines if a running task should be suspended upon another task reaching ready state
- This applies to CPU access, not access to I/O resources

Blocking

- One task indirectly blocks another by holding onto a needed resource
- Ex. Task A and B with
  - Assume
    - Priority A > Priority B
    - Both require resource R
    - Entry order is B, A
  - Sequence
    - B Reserves access to R
    - A enters and preempts B
    - A runs until it needs access to R
    - A must be suspended to resolve the problem to allow lower priority B to complete. This is B blocking A

Priority Inversion

- When blocking causes lower priority tasks to preempt higher priority tasks
- Example
  - Assume A,B,C with decreasing priority
- Scenario
  - C begins to run and reserves resource R
  - A enters and preempts C
  - A requests R, suspends to allow C to run and free R
  - B enters and preempts C before C is released
  - B runs until completion //This is the priority inversion as B preempts A
    //because A needs C to release a resource
  - C runs releasing R
  - A is resumed, uses R and completes
  - C is allowed to complete

More Vocab

- Turnaround time - the interval from the submission of a task to task completion
- Throughput - # of process completed per unit of time
- Response time - time delay from submission to the first action of task
- Waiting time - the time spent in "waiting" queues
- Example Queues that could be implemented:
  - Entry queue - has the tasks submitted but not yet ready to run
  - Ready queue - has tasks ready to run when CPU is available
  - I/O queue - has tasks waiting on I/o
  - Device queue - has tasks waiting for access to a particular I/O device

Scheduling Algorithms

- Asynchronous interrupt driven
  - Main program is a trivial infinite loop
  - Tasks are initiated by interrupts
  - The cpu can sit in low-power sleep mode waiting for events
  - Code outline →
  - Drawbacks
    - Difficult to analyze because it is based on external events

```
global variable declaration
ISR setup
function prototypes
void main(void){
    local variable declarations
    while (1); //task loop
}
ISR definitions
Function definitions
```

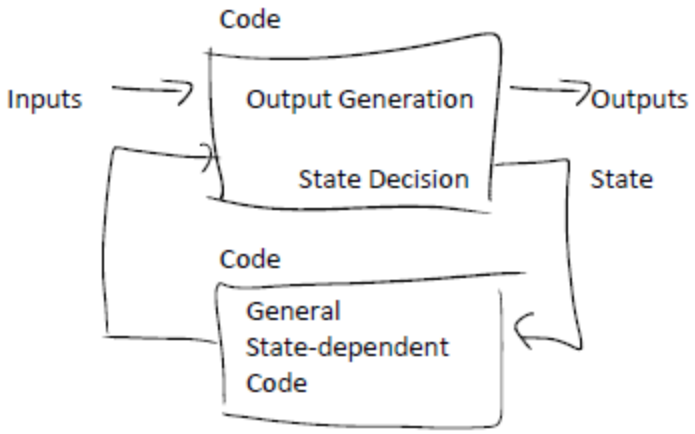
Scheduling Algorithms

- Polled and Polled with a Timing Element
  - Software polls signals and dispatches tasks
  - CPU is utilized more since it must check signals
  - Can add pauses to "align" events to steps of time (or sleep events to save power)

```
global variable declaration
function prototypes
void main(void){
    local variable declarations
    while (1){ //task loop
    /* can add pauses/waiting here if desired */
    test state of each poll signal
    an if, then,or switch construct to initiate tasks
    }
}
function definitions
```

Task Code Design

- State-Based Approach
  - For a state-based approach, a loop with two types of code are written
    - Some code is dedicated to processing inputs, setting output states, and deciding the next state
    - Other code performs operations based on the current state
  - Mainly event-driven, not always well suited for software especially if the number of states is large



Task Code Design

- Synchronous Interrupt Event Driven
  - Like asynchronous event driven, but event to trigger context switch decision is based on a regular timer
  - Timer triggers ISR to handle context switching and scheduling
  - Can implement time-sharing systems and implement preemptive scheduling (Using ISRs)

Task Code Design

- Combined Interrupt Driven
  - Allow context switch on regular timer events AND asynchronous input events
    - Make asynch ISRs short
- Foreground-Background
  - Mix of interrupt-driven and non-interrupt driven tasks
  - Interrupts signal foreground tasks while background tasks run

Time-Shared Systems

- Divide CPU time to multiple tasks
  - First-come first serve (non preemptive)
    - Task taken from front of queue and runs to completion
    - Not real time
  - Shortest job first
    - Each task has an associated expected CPU time before completion
    - Non-preemptive – tasks finish and next shortest is selected
    - Preemptive – if current task has longer to complete than another task it may be suspended
  - Round Robin
    - Blocks of time called “quantums” or “slices” are allocated in a rotating pattern to tasks
    - Task that don’t finish in given slice are suspended and moved to back of queue

Priority Scheduling

- Allow scheduling dependent on priority of task
- "shortest job first" is one example of priority scheduling
- With priority scheduling there can be issues:
  - (indefinite) blocking, priority inversion, starvation(some tasks never get a chance to run in a reasonable time, causing severe delays)
- Tasks can be prioritized based on
  - Execution time
  - Period allowed or deadlines
- Priority can be assigned upon submission or changed as conditions evolve
  - details in 12.3.8, we'll put off for later as time allows

Real-Time Scheduling

- Hard-real time
  - scheduler accepts tasks it knows it can complete in a give time
    - requires advance information about tasks such as execution time and I/O resources required so that reservations for resources can be made
    - works with I/O devices with well-defined timing
- Soft-real time
  - focuses on prioritization and low-latency dispatch, usually requiring preemption

Inter-Task Communication

- Considerations
  - What data is to be shared
  - Where data is stored
  - How to coordinate usage between tasks

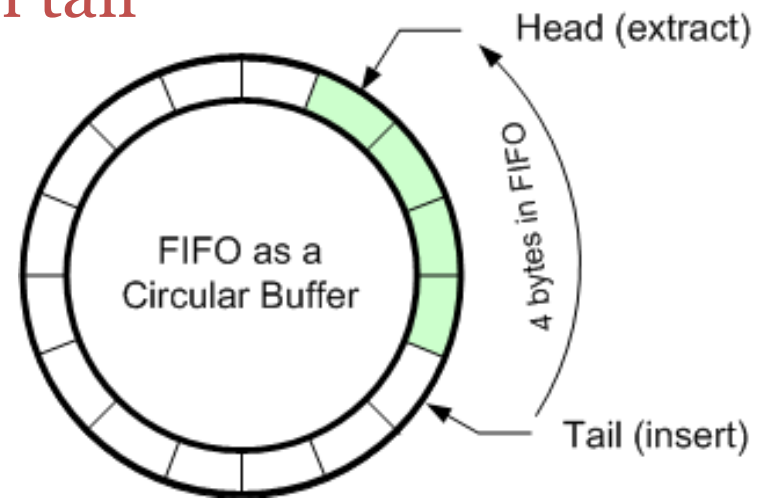
- Examples of data
  - A number (i.e. counter)
  - An Array
  - Status about a resource
  - Synchronization signals (i.e. mutex)

Shared Variables

- Global Variables
  - Need to coordinate access
  - Efficient for space and speed, important for embedded systems
- Shared Buffers
  - Can be a stack, FIFO, or other buffer type
  - Producers place data, consumer access and remove data
    - Needs methods to coordinate (isFull, isEmpty)
  - Overrun – when a producer fills a buffer faster than a consumer remove it (lose datapoints)
  - Underrun – When a consumer is left without any data

Ring (Circular) Buffer

- Ring Buffer – Data structure using an array and 2 pointers (head and tail)
  - When either pointer is at end of array, advances back to start
  - Insert at head, remove from tail
  - If head catches up to tail
    - Buffer is full
  - If tail catches up to head
    - Buffer is empty



Mailbox

- Mailbox constructs are provided by a full-featured OS. The basic idea is that each process has an incoming message queue that it can access, with messages posted by other processes.
- One process can signal that it is waiting for a message (data) in the mailbox and, if nothing is available in the mailbox, it can be suspended until a message shows up
- A mailbox can have a message (data) posted to it by another process
- Typical Code Interface:
  - pend (mailbox, data)
  - post (mailbox, data)

Message-Based Approach

- Mailbox concept can be expanded, for instance, to support buffers or communication among devices on a board or larger network. General message boxes can be read and posted to by multiple processes.
- Direct Communication -sender identifiers receiver and receiver identifies sender, or at lease sender identifies receiver
  - Send (T1, message) //send message to task T1
  - Retrieve (T0, message) //receive message from T0
- Indirect communication
  - Senders and receivers identify a mailbox
  - Send (M0, message)
  - Receive(M0, message)

Message Buffering

- One consideration is if link guarantees message order
- Regarding capacity there are 3 options:
- Zero-capacity link
  - Sender waits on receiver (so order is certain)
  - Execution timing tied together
- Bounded-capacity link
  - Sender may burst data, but can only send if buffer not full
  - Receiver must match sender rate "on average"
- Unbounded
  - No waiting, requires infinite storage or be sure that receiver removes data quickly enough

Access to Shared Resources

- Example using a shared buffer

Producer T0

```
int in  = 0;
while(1){
  while(count == MAXSIZE);
  B0[in] = nextT0;
  in = (in+1)  % MAXSIZE;
  count++;
}
```

Consumer T1

```
int out = 0;
while(1){
  while(count == 0);
  nextT1 = B0[out];
  out  = (out+1) % MAXSIZE;
  count--;
}
```

- Count read-and-modify operations should be atomic
  - Why?

Atomic Access using Flags

- Once the critical section of code has been identified, it can be protected with statements that
  - Check and flag use of resource, entry section
  - Flag the release of the resource, exit section

Producer T0

```
int in  = 0;
while(1){
  while(count == MAXSIZE);
  B0[in] = nextT0;
  in = (in+1)  % MAXSIZE;
  await(!T1Flag){
     T0Flag = true;}
  count++;
  T0Flag = false;
}
```

Consumer T1

```
int out = 0;
while(1){
  while(count == 0);
  nextT1 = B0[out];
  out  = (out+1) % MAXSIZE;
  await(!T0Flag){
        T1Flag = true;}
  count--;
  T1Flag = false;
}
```

Critical Section

- Must ensure “mutual exclusion” in the critical regions
  - i.e. turn off interrupts
- Prevent deadlock if two tasks are trying to enter their critical sections simultaneously
- Ensure ability to progress into critical section
- Bounded waiting
  - Limit the number of times a low priority task can be blocked by higher priority processes

Tokens and Token Passing

- Before accessing a resource, a process must acquire a “token”
  - Token is represented by flag or variable
  - Token is passed directly from task to task, initiated by task and not OS
- Problems
  - Some task can hold token forever
  - Task with token crashes (freezes)
  - Token is lost (communication problem)
  - Process with token exits without passing token
  - Managing queue of process to receive token
- Solution
  - System-level token manager that can recall and issue new tokens after time expires
- Tokens require a lot of communication and overhead

Interrupts

- Processes can disable interrupts during critical sections and prevent preempting
- A process that hangs in critical section can cause problems… like token passing problem
- A solution is to disable interrupts below some priority level, allowing a time-out interrupt to recover the system
- Not useful for processes running on different processors

Semaphores (Mutex)

- Concept:
  - Use variable to signal the locking of a resource
  - Simplest version involves a binary variable accessed through two ATOMIC functions

```
s is initialized to false in the system

 wait(s){
   while(s);
   s = true;
 }
Task T0
{
  …
  wait(s)
  critical section
  signal(s)
  …
}

 signal(s){
   s = false;
 }

Task T1
{
  …
  wait(s)
  critical section
  signal(s)
  …
}
```

Process Synchronization

- Aside from managing access to resources, semaphores can also be used to synchronize processes

```
Task T0
{
   …
   get input from user
        and put in a buffer
   signal(sync)
   …
}
```

```
Task T1
{
   …
   wait(sync)
   process user data
      in the buffer
   …
}
```

Extending Semaphores

- Rather than have a of process in a "spin lock" where it uses CPU cycles to check the synchronization signal until the instant the look is gone, you could sacrifice response time and free the CPU by suspending the process and putting it in an (OS-managed) queue to be woken when another process calls the signal function
- Semaphores can take on more than a binary value; this is useful for managing a pool of identical recourses and multiple processes accessing them

Monitors

- Semaphores represent a fundamental, low-level mechanism for resource locking and process synchronization.
- In general, tasks can use semaphores or you can write a "library" to access a particular resource that itself utilizes semaphores. For instance, imagine if a write command had a semaphore built into it to implement write blocking to prevent more than one process from writing at the same time.
- Monitors - another abstraction whereby access to a resource such as a buffer is managed by a abstract data structure with a public interface (functions) to access the resource and allow process sleeping and resuming based on conditions.

Starvation and Deadlocks

- Be aware of starvation and deadlocks when using semaphores.
- Starvation is when a process never has an opportunity to run and complete (or has to wait a very long time). This can be caused by LIFO queues.
- Deadlocks occur when a series of locks can be set up that prevent tasks from ever completing.

Deadlock

- Necessary conditions:
  - Mutual Exclusion-once a process has been allocated a resource, it has exclusive access to it
  - No Preemption-resources can not be preempted
  - Hold and Wait-some processes are holding one resource and waiting on another resource
  - Circular Wait-a set of processes exists in a state such that each process is waiting on a resource held by another in that set
- Example:
  - To make a peanut butter sandwich, bread, PB, and knife are needed. Assume two uncooperative children Suzy and Jonny run to the kitchen: Jonny grabs the PB and Suzy grabs the knife.
- Example:
  - variables representing back accounts actA and actB
  - To wants to initiate a transfer, so it reserves access to actA and is suspended
  - T1 wants to initiate a transfer from actB to actA, so reserves access to actB and waits for access to actA then suspends
  - To resumes and waits for access to actB

Handling Deadlocks

- Two basic ideas:
- Deadlock prevention
  - OS can monitor a deadlock condition or conditions and prevent at least one of them
- Deadlock avoidance
  - The OS can be given information about resources required by the task and delay tasks that may cause deadlocks. A hard real-time system requires this information

Deadlock Causes and Mitigation

- Mutual Exclusion
  - Avoid locking access where not required. Example: though a buffer may not be written by more than one process, read access may be shareable
- Hold and Wait
  - Don't allow processes to wait for resources if it already has some reserved. This requires a process to wait for and reserve all required resources to be available at once. Also, if a process requires an additional resource, it must first free all of its resources without condition. This may lead to low resource utilization and starvation of low-priority tasks.

Deadlock Causes and Mitigation

- No preemption
  - Any process that hold resources and requests another that is not available must allow its resources to be freed and if this happens a list of required resources is created and the process will resume when all of them are available …unless that resource being requested is held by another process that is itself waiting on other resources, in that case force that other waiting process to give up the resources.
- Circular Wait
  - Assign an order to all resources and enforce that no resources may wait on a higher-numbered resource while holding a lower numbered resource while order for requests. This requires requesting resources in-order and/or free resources lower-number resources when requesting a higher-numbered resource. In bank example, imaging if T1 waited for actA before asking for actB