# Intro

Operating Systems

* Software in between the user and computer hardware
* Coordinates access to hardware devices

Real-Time System

* Computer must respond rapidly as required by the user or the controlled process
* Applications are time-critical (WCET met); reliable; predictable; maintainable
* Maps input (from sensors) to output (by actuators)
* Hard vs firm vs soft real-time systems
  + Measure of extent of system failure when deadline is not met

# I/O Interfacing

Analog to Digital Converters (ADC)

* Analog voltage converted to binary representation
* Quantization and encoding
  + Resolution = (how large each ‘step’ of voltage is)
  + Signal is divided into 2b different quantization levels
* Nyquist sampling rate
  + A signal must be sampled at least twice as fast as its highest freq content

Software interfacing

* Isolated I/O
  + I/O registers accessed through a dedicated I/O bus between processor (CPU) and I/O device registers, special instructions to read and write
  + Separated memory and I/O address spaces
  + Instructions: IN, OUT
* Memory-mapped I/O
  + No dedicated bus or instructions
  + I/O device registers treated as memory locations; shared address space
  + Instructions: LDR, STR
* Arduino platform
  + 32KB flash (non-volatile) memory (for programs)
  + 1KB byte-addressable (for variables in prog execution)
  + 2KB SRAM (written and read from during program execution)
  + Analog inputs: A0 to A5 (ADC inside)
  + Analog outputs: marked with ~ (PWM output)

# Interrupts and Direct Memory Access (DMA)

Polling

* Amount of time and power wasted
  + Max incoming byte rate => period between 2 consecutive bytes
  + No. of status reads within that period = period/time needed to access device register
  + Power wasted = 1/no. of status reads

Interrupts

* Device controllers that move data to/from local buffer and to/from memory
* Inform CPU of finishing its operation by causing an interrupt
* Sequence of events
  + When interrupt becomes next task, save return address, fetch interrupt address
  + After executing ISR, go back to return address and continue
* Design issues
  + Note: Handling multiple interrupts
    - Assign priority, taken care of by programmable interrupt controller (PIC)
  + Problem: Interrupt response time = interrupt latency + processing time
    - Interval between interrupt raised and interrupt execution
    - Interrupt response time should be < period
  + Solution: Length of ISR code
    - Affects processing time as well as latency of lower priority interrupts
    - Affects delay of important tasks
  + Solution: Split interrupt processing
    - Perform a minimal amount of work in ISR, let a dedicated task complete main processing
* Memory management
  + Address ‘0’ stores reset vector: the first thing that runs
  + Reset vector will hold the address of ROM location
  + First few addresses will hold reset vectors/interrupts

Direct Memory Access (DMA)

* Used for high speed I/O devices
* Device controller (hardware) transfers a block of data independently from CPU, from buffer storage directly to main memory
  + Takes control of system buses either in burst mode (send all in one shot), or in cycle-stealing mode (CPU and DMA controller take turns to use system buses), or in transparent mode (send when the CPU is not using the system buses)
* One interrupt generated when done per block of data
* DMA not supported by Arduino

# Real-Time Software Architectures

Simple Polled Loop

* Testing/polling a flag for an event occurrence
* Add delay to account for contact bounce (that may set the flag multiple times)
* Pros:
  + Fast response; mainly for low-end systems
* Cons:
  + Fails in cases of event bursts
  + Not sufficient to handle complex systems
  + Wastes CPU time

Round-robin (without/with interrupts)

* A set of n self-contained tasks in a continuous loop
* Can set the same or multiple cycle rates by repeating tasks in the cycle
* Pros:
  + Works well when there are few devices and frequency is the same
  + Tasks are short, no tight time requirements
* Cons:
  + Fails when any device has a shorter cycle than the time taken for one loop
  + Fragile, might fail when a single device is added
* Adding interrupts:
  + ISR for time-critical processing
  + Main loop polls the flags, handles longer running code
  + Pros: Offers greater flexibility, devices get attended to immediately
  + Cons: May still take a while for data to be processed

Function queue scheduling

* Interrupts that insert a pointer to a function to process the data
* Pros:
  + Easy to enforce priorities by managing the queue
* Cons:
  + Added complexity from function queue

Real-time OS

* Can handle multiple tasks, facilitate inter-task communication and synchronization, manage memory, etc
* Useful when applications involve many parts that interact
* Pros:
  + Clean, convenient and predictable way to control complex applications
* Cons:
  + Relatively huge, must be customized, complex

# Task Management

Memory Management in a Single Process

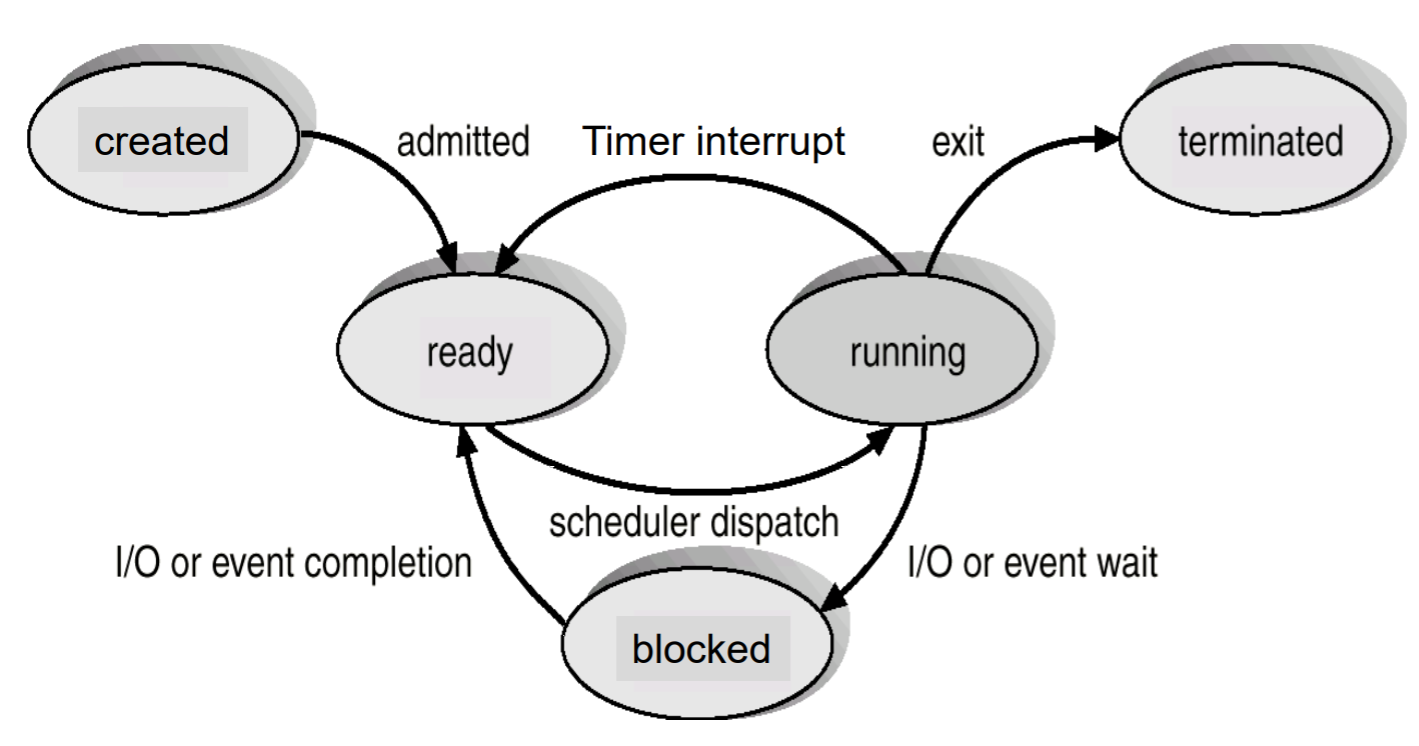
* Stack (parameters and local variables)
* Heap (dynamically allocated variables)
* Data (global parameters)
* Text (code section, program)

Process Control Block (PCB)

* Each process has its own PCB stored in main memory
* Keeps information associated with each process
* Pointer, process state, process ID, program counter, CPU registers, memory management info, info regarding open files

Process State/Context

* States: Created, running, blocked, ready, terminated



* State transition from running to ready:
  + Pre-emption (involuntary)
    - OS scheduler forces running process to relinquish processer
  + Yield (voluntary)
    - Process asks OS to reconsider the scheduling decision
* State transition from running to blocked
  + Waiting for completion of I/O; synchronizing and communicating with other processes
* State: Terminated
  + When process can no longer be executed
  + Can be spontaneous or due to an error
  + Process and its PCB eventually removed from the system

Context Switch

* When CPU switches from one process to another, system must save the old process context
* Overhead: system does no useful work while switching
* Scheduler – the component that decides which process is executed at any given time

Threads

* Multi-threading: multiple threads within a single process
* Thread control block (TCB): thread ID, program counter, register set, stack space
* Shares with threads in the same process: code section, data section, OS resources

# Real-Time Scheduling

Intro

* Terminology
  + Worse Case Execution Time (WCET): C
    - Execution time without interference from other activities
  + Worse Case Response Time (WRET): R
  + Period (P)
  + Deadline (D)
  + Interference (I): sum of computational demands from all higher priority tasks
  + Feasible schedule: schedule meets all timing constraints
  + Optimal scheduling algorithm: an algorithm that produces a feasible schedule
* Assume task independence and negligible overhead time

Rate Monotonic Scheduling (RMS)

* Optimal fixed-priority scheduling
  + RMS guaranteed to pick out an optimal fixed priority schedule if one exists
* Priority inversely proportional to period; independent of WCET
* Hyper-period: LCM of task periods
  + Schedule repeats every hyper-period
  + If feasible for one hyper-period, then task set is schedulable
* Schedulability
  + Utilization: U =
    - Fraction of processor time spent in execution
  + U 1 : Necessary condition i.e. feasible schedule *may* exist
  + U : Sufficient condition i.e. feasible schedule will exist
* Rate Monotonic Analysis (RMA)
  + When U 1
  + Critical instant: the instant when the release of the task produces its largest WCRT
    - When a task is released simultaneously with all higher priority tasks
  + For task to be schedulable, C + I D
  + Schedulability test:
    - At the instant before any new task is released, check if total demand capacity (total demand = demand of tasks that have been released so far at that moment)
    - Both a necessary and sufficient condition
    - Formula: A picture containing object

      Description generated with very high confidence

Earliest Deadline First (EDF) Scheduling

* Optimal dynamic priority scheduling scheme
* At any instant, task with earliest deadline has the highest priority
  + Priority may be updated only when a new task instance is released
* Necessary and sufficient condition to produce a feasible schedule: U 1

Cyclic Executive aka timeline/cyclic scheduling

* Creates an offline, static schedule for the hyper-period that meets all deadlines
* Major cycle: LCM of tasks (hyper-period)
* Minor cycle: GCD of tasks (all periods must be an integer multiple of minor cycle)
  + Synchronization points; execution switches between minor cycles at periodic timer interrupts; tasks are activated in sequence
* Pros
  + Minimizes pre-emptions if constructed correctly
  + All procedures share the same address space
  + No current access, so no need to protect shared data
  + Straightforward implementation: no scheduling and context switching overhead
* Cons
  + Handling one task with disproportionately large period
    - Ignore that task first
    - Then, if task’s period is a multiple of major cycle, handle it with a secondary schedule (call wrapper code once every major cycle, invoke every n calls)
    - Wasteful compared to constructing a long major cycle schedule
    - Some task sets may become non-schedulable because of this
  + Task must be split if WCET > minor cycle
    - May be impossible to find a point in code where a split will meet timing requirements
    - May be concurrent data access across tasks if split badly