08/30/21

- Dates
    - Oct 4: 1 page project proposal due
    - Dec 1: Exam (25%)
    - Dec 8: Posters/demos (30%)
    - Dec 15 4pm: Project report due (20%)
    - Weekly homeworks (15%)
- Readings
    - Real-Time Systems K Shin's book
    - IEEE RTSS
    - IEEE RTAS
    - ACM/IEEE ICCPS
    - International Journal of Time-Critical Computing
    - ACM Transactions on Embedded Systems
    - ACM Transactions on Cyber-Physical Systems
- Homework
    - Read and analyze 2 or more recent papers on topics covered during the assignment period (4 page long report including references)
        - Cover page
            - Title of topic, name, e-mail address, date of submission, and brief summary of articles read
        - Analysis and critiques
            - Critically analyzed
        - If I were the author, what would I do differently?
        - References
- Term Projects
    - Team of up to 3 total members
        - Can use project for research but not as another class
        - Literature surveys or slight modifications of existing work not allowed
        - Should be publishable
- Notes
    - Trade-offs apply to everything including airplanes, embedded systems, AI
        - Efficiency, robustness, usability, security, speed,
    - Do a paper presentation
    - 5 days to regrade on anything
    - Research is defined as creation from nothing or from ill-conceived notions
    - Finish PhD feeling like you can do anything
- Class Content
    - Real-time systems may be defined by particular granularity of time (ms, s) it needs to be in before it fails
        - Deadlines can come from law of physics or can be artificially imposed
        - Soft real-time system is where user is unhappy if not done by a deadline

        - Hard real-time system is where system doesn't work at all if not done by a deadline

09/01/21

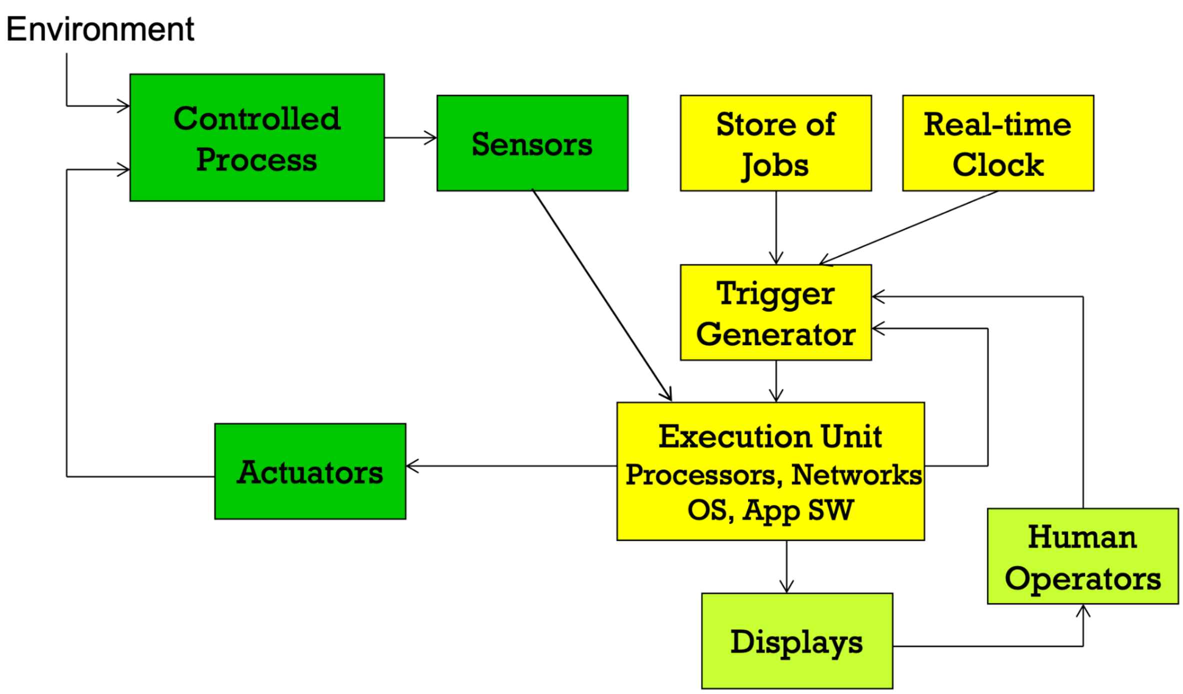
- How fast can you acquire data, process it, and actuate decisions?
    - Achieve all 3 steps before the deadline
    - For example, cars traveling fast may not be able to stop/react as fast, also depends on road conditions
    - Deadline's could be random variables as well (or noisy to some degree)
    - We digitize/discretize analog signals at a specific frequency (sample interval)
        - Sampling theory
        - Sample more as car goes faster
    - End to end latency is also considered application latency
        - How to allocate deadline time to individual components? (deadline distribution)
- Artificial deadlines created from usability studies
    - Provide safety margins where you have to miss many deadlines before failure
- The same task could be hard real-time or soft real-time depending on the state of the system
    - Tasks/messages/packets may be triggered periodically, aperiodically or sporadically (2 consecutive instances must be infrequent to some minimum)
    - Braking is a sporadic task, combined with detecting an obstacle which is periodic
- Typically assume 2 consecutive failures take longer than the recovery time
    - Multiple failure before recovery can cause issues
    - Lump multiple simultaneous failures as a single failure
- We want to optimize and create adaptive schedules because:
    - Computation takes time, generates heat, consumes energy, consumes bandwidth
- Requirements
    - Size, power (heat), weight, radiation/EM hardened
    - Performance must be responsive and predictable
    - Must be cheap and short time-to-market
    - Must be safe, reliable, secure/private

09/03/21

- System state can be handled by external triggers via polling or ISR (interrupt handler)
    - Interrupt done between instructions not in the middle of instruction execution (time consuming [flush cache, save registers, etc.])
    - Polling (spinlocks) [better when it'll clear up soon]
    - Event and/or time-driven state transitions
        - From input to output, you have to go through a series of states
        - State can also be considered with the number of processors

        - Or CPU is in WAIT, EXECUTE, SUSPEND
        - Event driven (conditional), time driven (every n seconds)
- Timing constraints and multi-threading
    - Given x at time t1, produce y by t2
    - Non-deterministic, race conditions, time-dependent behavior, etc.
    - Failures are rooted in interaction of multiple concurrent operations and threads
- RTOS
    - Use host and target systems
    - Needs to be a good resource manager

A Typical Real-Time Embedded System



Green is controlled processes, yellow is the controller
- You can model a lot of things in this manner (humans, cars, internet, etc.)
- Process keeps cycling until mission is complete

<u>09/08/21</u>

- Trends
    - Proliferation
        - Industrial, RFIDs, sensor networks and ad hoc wireless, medical, smart spaces and assisted living
    - Integration at scale
        - Low end
            - Sensor networks, world wide sensor web

            - Ubiquitous embedded devices, large scale networked embedded systems, seamless integration with a physical environment
        - High end
            - Power grids, navy ships, global information grid
            - Complex systems with global integration
    - Biological evolution
        - Exponential proliferation of embedded devices (Moore's law) is not matched by an increase in human ability to consume information
        - Increasing autonomy (human out of the loop)
    - These trends all come together to a distributed cyber-physical information distillation and control systems (of embedded devices)
- Electric Vehicles as an example
    - Components are all independent so turning off the car doesn't turn off parts
    - Power system in EVs
        - Powertrain, AC, radios, window lift, sunroof control (must need communication and control)
        - Cyber physical coupling. There should be cyber capabilities in every physical components (large scale wired and wireless networking)
        - System of systems has spatial-temporal constraints (dynamically reorganizing/reconfiguring)
        - Also has security and privacy needs
        - Control loops keep looping (must close loop, example loop time 1 ms)
            - High automation
- Electric power grids
    - Equipment protection devices trip reactively and locally
    - Cascading failure (2003)
    - Real-time cooperative control of protection devices
    - Self healing islands of stable bulk power
    - Issue: conventional operational control concerns for bulk power stability and quality, flow control, and fault isolation
    - Context: market behavior, power routing transactions, regulations
    - Disposing extra electricity is non-trivial
- Health care and medicine
    - Medical records at any location
    - Pulse oximeters, blood glucose monitors, insulin, fall detection
    - Operating room should be closed loop monitoring and control, plug and play, robotic microsurgery

<u>09/10/21</u>

- Sporadic tasks
    - Hard deadline
    - Highly critical task
    - Executed whenever there's time

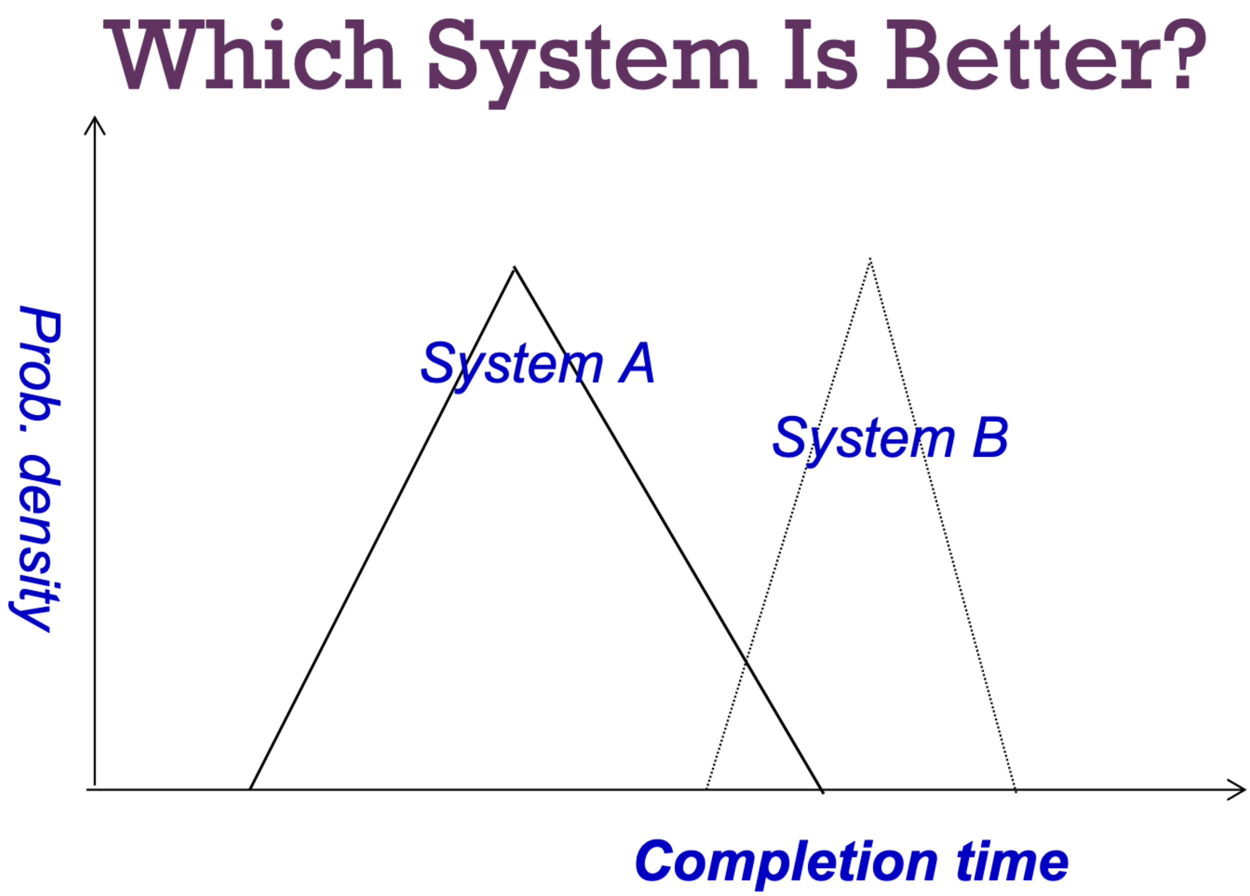
        - Rejected by scheduler if there's less slack time
        - Deadlines are met easily
    - Aperiodic tasks
        - Soft deadline
        - Low or moderate critical task
        - Execution doesn't depend on available slack time
        - Never rejected by scheduler
        - Meeting all deadlines is difficult
    - Each task has a priority depending on the scheduler

<u>09/13/21 - CPS</u>

- Grand visions
    - Near-0 automotive traffic fatalities, minimal injuries, reduced traffic congestion and delays
    - Blackout-free electricity
    - Perpetual life assistants
    - Extreme-yield agriculture
    - Energy-aware buildings
    - Location-independent access to world-class medicine
    - Physical critical infrastructure that calls for preventive maintenance
    - Self-correcting and self-certifying cyber-physical systems
    - Reduce testing and integration time of complex CPS
- Example: Battery awareness
    - Don't want to overcharge battery
- Potential accidents
    - Unsound interconnections
    - Feature interactions that are unanticipated
    - Inadequate development infrastructure
    - System instabilities
- Interaction modes
    - Computation resources
    - Shared resources
    - Controlled plant
    - Human operators
    - Larger environment
- Formal methods
    - Instead of testing or simulation, uses automated model checking, theorem proving, static analysis, run-time verification
    - Exponential complexity:
        - Best when property is simple or system is small/abstract
        - Model rather than C-code

<u>09/15/21</u>

- Characterizing RTES
    - How to measure “goodness” of RTES?
    - How to estimate exec time of a program given source code and target architecture?



-
    - Execution time
        - A
    - Predictability
        - B
    - aM+bV, or (M, V)
        - weighted sum of mean response time and variance
    - How do we rank the two?
- How to measure RTES performance
    - MIPS?
        - No, depends on architecture (RISC = 1/1.2 clock cycles) (CISC = 1/1.8 clock cycles)
    - Want RTS performance measure to:
        - Be efficient encoding of relevant information
        - Be objective means for ranking candidate systems for an application
        - Represent verifiable facts

- Performance measures
    - Reliability: R(t)
    - Availability: A(t)
    - Throughput
    - Capacity reliability
        - Probability of not being in any failure states
    - Computational reliability
        - Probability system can start task T at time t and in state s
    - Performability
        - Given n accomplishment levels, performability is where probability the computer functions to allow the controlled process to reach accomplishment
        - A1, A2, …, An and P(A1), P(A2), …, P(An)
        - Hierarchical format
            - Accomplishment levels
            - Accomplishment of controlled-process tasks
            - Capacity of RTES to execute specified algs for control tasks
            - HW structure, OS, application SW
- Cost functions and hard deadlines
    - Hard deadlines are the maximum controller “think” time that will allow controlled process to be kept in a stable state space
    - Cost of the response time C(r) = P(r) - P(0)
        - Where P(r) = performability associated with response time r
    - Hard deadline keeps deviations within a specified bound
- Task execution times
    - Depends on source code, compiler, machine architecture, OS
    - Need an ideal tool which takes in all these factors and outputs a task execution time
    - Analyze straight-line source code
        - Estimate execution time of each microinstruction
        - What about loops and conditional branches?
            - Depends on input data, interrupts,
    - Difficult to estimate task execution time
        - Difficult to determine # times an instruction will be executed
        - Time to execute instructions is not constant
            - Depends on pipelining, out of order execution, cache, branch prediction, multiple instructions per clock cycle, multiple cores on a single die
        - Instruction execution time depends on instruction, data, and state of machine
- Modeling concurrent task execution in a distributed real-time control computer system

<u>09/17/21</u>

- Execution time analysis
    - Hard real-time constraints/deadlines
    - Soft real-time constraints/deadlines
    - No set execution deadline for a given task
    - Is there a run i for which t_run,i > t_d?
        - Worst case execution time analysis
- Path analysis
    - for (i=0; i<100; i++) {
      ```
              if rand() > 0.5
                      j++;
              else
                      k++;
      }
      ```
    - 2^100 feasible paths
        - Cannot enumerate all possible paths
        - Analytical approach required
- Count analysis
    - Basic block
        - Sequence of instructions which are all executed if the 1st one in the sequence is executed
        - Block with no branches or loops
    - Steps
        - Divide program into basic blocks
        - Determine execution time of each block
        - Determine possible number of executions for each basic block
        - Maximize sum of execution time * # executions for each basic block
    - ```
      x1      k=0;
      x2      while(k<10){
      x3              if(ok)
      x4                      j++;
      x5              else{
                              j=0;
                              ok=true;
                      }
      x6              k++;
      }
      ```
    - Can design a control flow graph (CFG) to draw a graph from code
        - Draw arrows for where the code goes, for each block as a node
- Integer linear programming formulation
    - Structural and logical constraints build a set of equations
    - Maximize sum obeying to all constraints
    - Objective function is linear and all constraints are linear expressions
        - ILP solver is guaranteed to determine the extreme case solution
      ```

        - Chronos
- ILP techniques for caches
    - Memory hierarchy pyramid (processor -> registers -> caches -> RAM, main memory)
        - Cache hits / cache misses
            - 2 different execution times
            - c^{hit}, c^{miss}
            - x^{hit}, x^{miss}
        - Sum of $c^{hit} * x^{hit} + c^{miss} * x^{miss}$
    - Assume direct mapped caches
- Line blocks
    - Basic blocks can content several instructions mapped to different cache lines
        - Would have to grab memory from different cache lines
        - Execution times differ depending on program structure
    - Contiguous sequence of code within same basic block that's mapped to the same cache line in the instruction cache
    - $B_{4,1}$ $B_{4,2}$
- Basic blocks to line blocks
    - Draw a table, look at number of cache sets
    - 0, 1, 2, 3
    - B_1 and B_3, B_1 and B_3, B_1 and B_2, B_2
    - Can group together 0 and 1 because their blocks are the same
    - Whenever you hit a line block for the 1st time, it'll always result in a miss
    - Any 2 l-blocks that map onto the same cache set are called conflicting if they have different address tags
        - 2 non-conflicting l-blocks are mapped to the same cache line
    - Sum of basic blocks (sum of line blocks)
        - $c^{hit} * x^{hit} + c^{miss} * x^{miss}$
- Cache conflict graph
    - For each cache set containing 2 or more conflicting l-blocks
        - Start node, end node, and node $B_{k.l}$ for every l-block in the cache set
    - Edge from $B_{k.l}$ to $B_{m.n}$: control can pass between them without passing through any other l-blocks of the same cache set
    - Start node, end node
        - Put nodes for each line block in the cache set

09/20/21

- Pipelining and caches
    - Fetch -> decode -> operand fetch -> execute -> result store
    - 5 concurrent instructions in execution
    - Timing complexity because of data inter-dependencies, branches, interrupts
    - Caches fix speed disparity b/w CPU and memory
    - Smarter cache avoids misses, divide into exclusive and shared areas

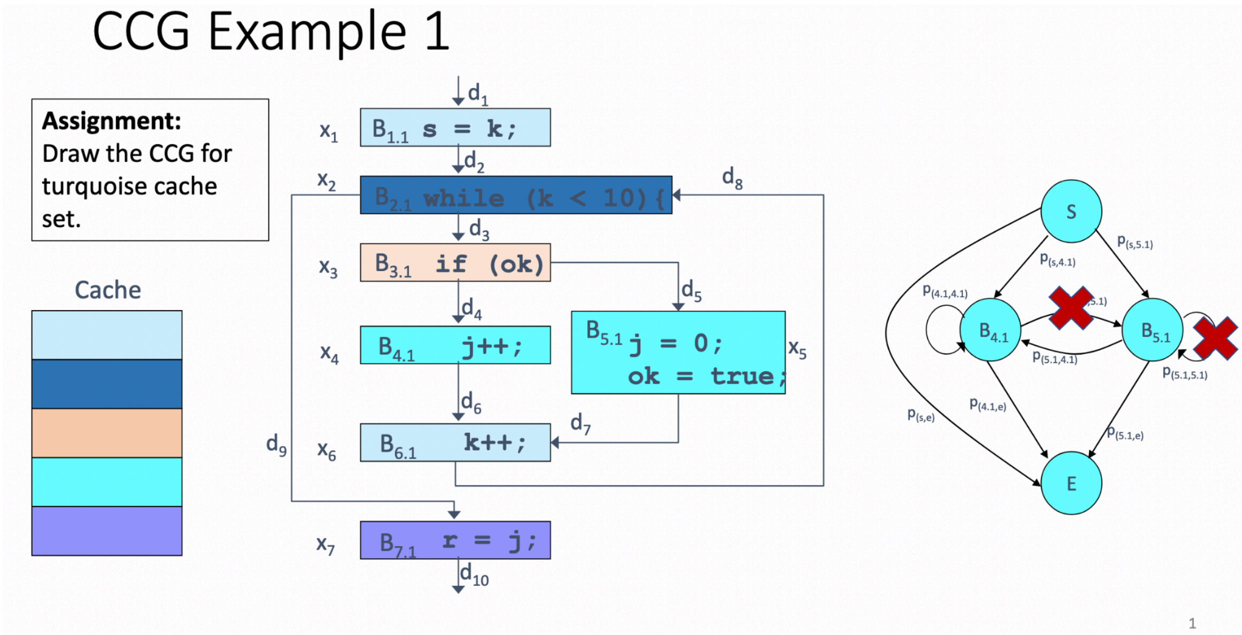
        - What about virtual memory for real-time systems?
            - Page faults (item isn't in memory, have to fetch from disk)
        - Control speculation (branch prediction)
- Execution time of concurrent tasks
    - Many CPS and RTES require multiple dependent tasks to run concurrently (not just single threaded)
    - Need to model concurrent tasks for their execution times and scheduling
    - Model must simultaneously consider both processing architecture (platform) and tasks (application)
- System model
    - Platform architecture
        - Processing node architecture, registers, pipelines, caches
        - Operating system
        - Networking protocols
    - Task system
        - Application
        - Assignment (tasks)
        - Scheduling (modules or activities)
        - Activities are modeled by Generalized Stochastic Petri Nets (GSPN) which are converted to Continuous-Time Markov Chains (CTMC)
            - Markov chain -> math -> execution time prediction
        - Precedence constraints on tasks
            - Key to capturing dependencies between tasks
- Application modeling
    - Task-oriented: too coarse to capture details
    - Module-oriented: difficult to study
        - Message scheduling policies
        - Communication protocols
        - Task execution stage of each PN
    - Approach
        - Contiguous stretches of code are combined into activities without losing precedence constraints and avg/worst execution times
        - GSPN -> sequence of CTMCs to model task system evolution
        - Task flow graph
            - Chain, AND-FORK & AND-JOIN, OR-FORK & OR-JOIN, Loop
                - OR doesn't wait for late branches, AND does wait
            - Can construct any program using these 4 components
            - Can build a task tree to describe this task flow graph (TFG) with 4 subgraphs
- Definitions
    - Module: combination of 2 or more code stretches or modules
    - Activity: largest module that can be formed without violating precedence constraints
    - Marked Petri Net: C = (P,T,I,O,*u*) where *u* : P->\# of tokens for place p in P

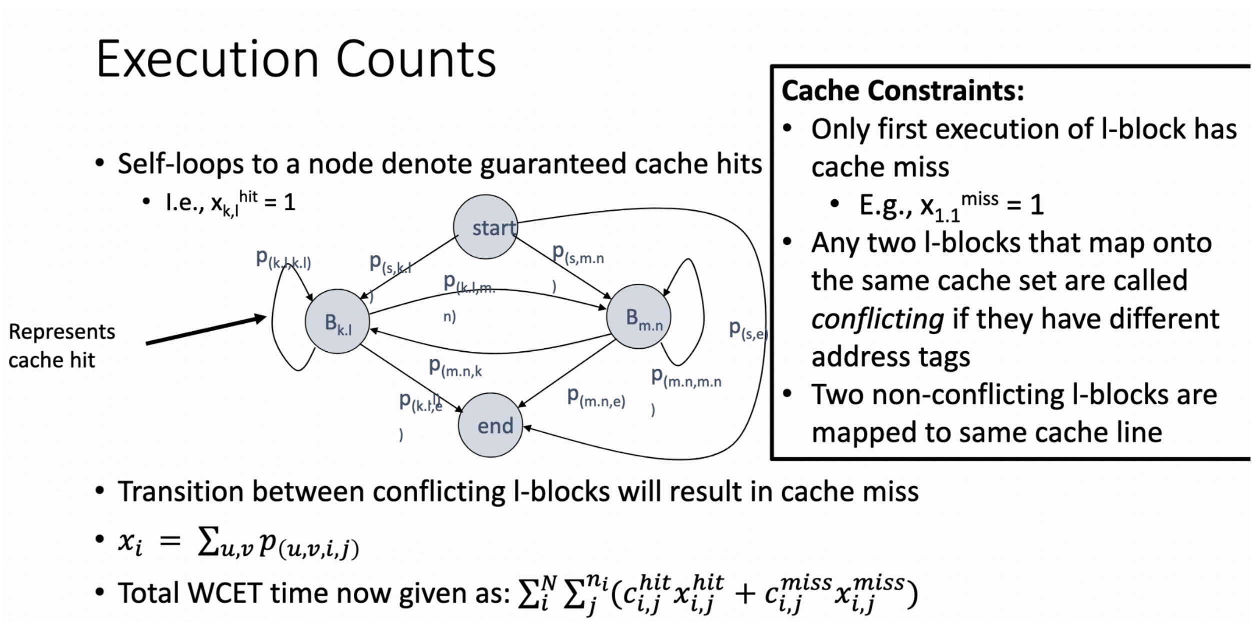
        - P is set of places
        - T is set of transitions
        - I is input
        - O is output
        - *u* is tokens (mapping indicating progress of execution [board game token])
    - GSPN: marked Petri Net with a nonnegative random firing delay for each transition t in T
        - Example: SEND-RECEIVE-REPLY, REQUEST-RESPONSE, WAITFOR
- Mert notes
    - Sum of control flows going into a node should be equal to the sum going out of a node
    - If you have self-loops in your control flow graph, it always represents a cache hit
    - If you have a transition between 2 conflicting I-blocks will always result in a cache miss
    - Read “Cache Modeling for Real-time Software: Beyond Direct Mapped Instruction Caches”

<u>09/22/21</u>

- Step through the GSPN model token-by-token
- There may be probabilities for each transition
- If there’s a deadlock in the GSPN, it’ll just timeout at (5ms) so no one cares
- Continuous time markov chain
    - If there’s and end state involved, it could be time-critical

<u>09/24/21</u>

- 

    - Look at each conditional branch and draw the flow
    - Label edges, nodes
- 

    - Transition between conflicting l-blocks will result in cache miss
    - Sum of edges going into it (or out of it)
    - Assume entire cache is empty before starting
    - c_ij is execution time of each line block, x_ij is number of executions of each line block
- CFG to identify l-blocks
- CCG to identify cache constraints

<u>09/27/21</u>

- Will real-time application really meet its timing constraints?
    - Feasible/optimal
    - Release time
    - (absolute, relative, effective) deadlines/release-times
    - Precedence relation
        - Set of tasks that must be completed before task T can begin its execution
    - Resource requirements
        - Processor, memory, bus, disk
        - Can either be exclusive or shared (read-only, read-write)
        - Schedule
            - Offline or online
                - Sometimes you don't know all data required for computational workload in advance
                - Examples could be interrupts or unexpected events
            - Sometimes priority is static or dynamic
            - Another task might preempt its execution (taking over priority of execution)
            - Uni-processor or multi-processor
    - More terminology
        - Hard deadline (late result has little/no value or leads to catastrophe)
        - Soft deadline (late result can still be useful)
        - Tardiness
            - Min(0, deadline - completion time
        - Utility
            - Function of tardiness
        - Release time
            - Could be a fixed release time or there could be jitter/noise (sporadic or aperiodic)
            - A job can be released later than that of its successor
        - Execution time
            - Unpredictable due to memory refresh, DMA, pipelining, cache misses, interrupts, OS overhead, execution path variations, etc.
        - WCET
            - A deterministic parameter for the worst case
            - Conservative measure, an assumption to make scheduling feasible
        - Job
        - Deadline of a job can be earlier than that of its predecessor
        - Effective release time = max(release time, effective release time of all predecessors)
        - Effective deadline = min(deadline, effective deadline of all successors)
            - These are recursive definitions (if no successor/predecessor, effective = deadline/release)

        - Rate monotonic (RM): statically assign higher priorities to tasks with smaller periods
        - Deadline monotonic (DM): the smaller the relative deadline, the higher the priority
        - Earliest deadline first (EDF): the earlier the deadline, the higher the priority (this is optimal if preemption is allowed and jobs don't contend for resources)
        - Maximum laxity first (MLF): the smaller the laxity, the higher the priority (also optimal)
            - Laxity is the laxness of your time to execute something (deadline - execution time) - right now

09/29/21

- Utilization is the fraction of execution time over the period
    - u = e/p
- High priority task should preempt the low priority tasks (priority inversion)
    -

10/01/21

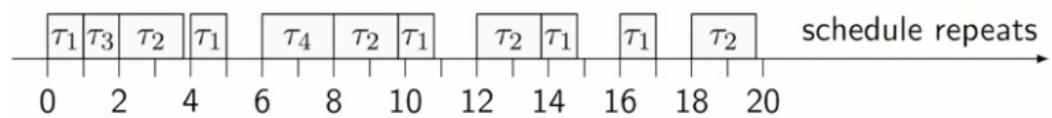
- Clock driven
    - Static or off-line scheduling (calculated a priori)
    - Decision is made at a priori at chosen time instants
    - Uses a hardware timer and no OS
    - Regularly spaced time instants
    - Schedule is computed off-line and stored for use at run-time
        - All parameters of hard real-time jobs must be fixed and known
        - Scheduling overhead during run time is minimal
        - Complexity of scheduling algorithm is not important
        - Good schedules can be found
        - Disadvantage: no flexibility
    - *n* periodic tasks, tau_1 to tau_n
        - Task is specified with phi, T, C, D (task phase, task period, execution time, deadline)
        - Shortened to T, C (period, execution time)
        - Only 1 processor
    - Schedule table
        - Occasionally CPU will be idle and no task is scheduled (x)
        -

	T1	T2	T3	T4
Period	4	5	20	20

Execution time	1	1.8	1	2

-

Time	0	1	2	3.8	4	5	6	8	9.8
Task	T1	T3	T2	x	T1	X	T4	T1	X

- 

- Only show for the Least Common Multiple of period (20)

- Frame-based scheduling
    - Problems: big number of tasks, big schedule table, embedded systems have limited memory, reprogramming timer might be slow
    - Idea
        - Divide time to constant-size frames
        - Combine multiple jobs to a single frame
        - Scheduling decisions made only at frame boundaries
    - Downsides
        - No preemption, each job must fit in frame, schedule calculation + various error conditions (task overrun)
    - *f* is frame size, how to select *f*?
        - Constraints
            - we want big enough frames to fit every job without preempting it
                - f >= max(C_i) for i=1,...,n
            - In order to have a small table, f should divide H. Since H = LCM(T_1,...,T_n), f divides T_i for at least one task T_i
            - Let F = H/f (F is integer).
            - H is called major cycle and f is minor cycle
        - We want frame size to be small so there is at least 1 frame between task release time and deadline
        - 2f - GCD(T_i, f) <= D_i
            - deadline for i
        - Summary
            - H = LCM (T_1,...,T_n)
            - f >= max(C_i) for i=1,...,n
            - f should divide H
            - 2f - GCD(T_i, f) <= D_i
        - How to find f?
            - Start with f >= max(execution time)
            - Then find f's that divides H
            - Finally, check each GCD equation value for each of these
- 3 tasks (in helicopter control system)

        - 180x per second, computation time 1 ms
        - 90x per second, computation time 3 ms
        - 30x per second, computation time 10 ms
- Hard real-time jobs have a hard deadline, doesn't matter if it's done early
- Always schedule aperiodic jobs at the beginning (interrupt based first)
- Slack stealing
    - There are periods of idle time on the CPU
    - Without slack stealing, do periodic hard tasks first and then fill in
    - With slack stealing, can move periodic hard tasks later and prioritize the aperiodic jobs
-

<u>10/04/21</u>

- 3 big questions
    - Where am I?
        - GPS + digital maps
    - Where to go?
        - Mission/route planning
    - What's around me?
        - 360 sensing
    - Sensor types
        - GPS, LIDAR, Images, CAN, WIFI/5G, Integrated Display, Ultrasonic Sensors, Full-operational Arch, Multicore, FPGA, FlexRay, Ethernet, DSRC
- Need to do the above with large volume, long operation time, uncertain operation environment, reliably and safely, mixed traffic
- SAE levels
    - 0: No automation
    - 1: Driver assistance
    - 2: Partial assistance
    - 3: Conditional automation
        - From here an above, any issues are human fault (human final decision)
    - 4: High automation
        - From here beyond, any issues are manufacturer's fault
    - 5: Full automation
- Most things we talk about are level 4+
- AV system components
    - Environment sensing -> perception and planning -> motion control and vehicle operation
    - Needs to be performant, safe, and affordable
- Approach for automated driving
    - L5 (gradually pull back to lower level)
        - Object detection

        - Multi LIDARS and multi cameras and detailed HD map
    - Working environment
        - Day + night with rain and snow
    - Image annotation
        - Semi-automatic with human assistance
    - Training technique
        - No pre-training/reinforced learning
    - Limitations
        - Works better on predefined routes
- L1/L2/L3/L4 (gradually grow to higher level)
    - Object detection
        - Single camera and multi radar sensors
    - Working environment
        - Daytime with bright light
    - Image annotation
        - Manual annotation by human
    - Training technique
        - Supervised training
    - Limitations
        - Fallback to human driver
- AV cost
    - $$$, space, driving range, warranty, maintenance
- Perception
    - Camera, LIDAR, Radar
        - Different advantages
            - Camera (Best sensor for color and texture interpretation)
            - LIDAR (High precision detection without light/sound interference)
            - Radar (Cost effective, good as backup sensor)
        - Different disadvantages
            - Camera (High processing required)
            - LIDAR (Needs HD map, requires huge amounts of data, expensive)
            - Radar (Poor resolution, 2D information only)
    - Processing
        - Algorithms
            - HOG, sobel, SVM
            - Alexnet, Squeezenet, SSD, YOLOv3
            - Optical flow, ORB
        - Timing characteristics
            - Constant execution time
            - Need to detect multiple at once
            - Tradeoff between processing delay and accuracy

- Some algorithms can do it fairly well with small amount of time, can possibly spend longer to compute higher accuracy result
      - Single frame processing delay - batch processing may be limited
      - Time synchronization among multiple sources
    - Hardware computing platform
      - Multicore, many-core, accelerators (DSP, GPU, FPGA)
  - Challenges
    - High performance, high computation, safe and secure, affordable, optimization with large number of parameters
    - Focus on vision
      - Vision processing are main components posing challenges
      - Lidar processing shares challenges (less computation load)
      - Radar is light computing workload
    - Deep learning / neural network for vision processing
      - Inference of pre-trained CNN
      - Special cases for adaptive learning or reinforcement learning
- Development process
  - Algorithm developed on machine -> portable across different computing platforms -> enable system level optimization and analyzability -> meet requirements on timing, safety, security
- Solution concepts
  - Objective: run a CNN inference algorithm effectively and efficiently
    - Computation reduction
      - Quantization (32bit -> 16bit) and pruning
      - Can binarize CNN (use bit operations instead of floating point precision)
      - Selective processing
        - Crop regions of interests
        - Use camera to guide LIDAR
    - Architecture optimization
      - Hardware acceleration
        - Multi-core CPU
        - GPU
        - FPGA
        - TPU
      - Parallel processing
    - Device sharing
      - Multiple vision applications using different resources
      - Need to synchronize using locks
- Preemptable CNN

    - Meaningful result only retrieved at the end of the process
    - Models get more complex with more layers
    - Easy to schedule if preemptable
    - Deal with different levels of CNN importance/criticality
    - Desired CNN with fine-grain execution control

<u>10/06/21</u>

- Schedules
  - Earliest deadline first (EDF) schedule
    - Preemptive dynamic priority scheduling
      - Job with earliest deadline has priority
    - Non-preemptive or multiple processors is non-optimal
  - Least slack time (LST)
    - Preemptive priority scheduling based on slack time (deadline - execution time)
    - Optimal for preemptive single processor schedule
- Schedule anomaly
  - The schedule fails even after we *reduce* job execution times
  - Preemptive is much easier than non-preemptive scheduling
- Aperiodic tasks
  - A periodic server follows the cyclic schedule and looks at aperiodic task queue
  - Slack stealing
    - Slack time is how much each periodic task can be delayed
    - Assume all tasks must be completed before the end of their frames and aperiodic tasks are not preemptable
    - Do slack stealing at beginning of each frame and examine queue when idle
- Scheduling goal
  - No deadlines missed for all jobs invoked by a set of periodic tasks
- Scheduling algorithm
  - Determines when to execute a task (EDF, RM)
- Schedulability analysis
  - Guarantee no deadline misses of a given task under a scheduling algorithm
- Real-time scheduling
- Assumptions
  - Single processor
  - Hard deadline
  - Independent periodic tasks
  - Relative deadline = period
  - Preemptable without any limit
  - No overhead for context switch
- Why should you start with a simple theoretical model?
  - Shannon for example started with a useless impractical simple model

        - After solving the simple model, he started adding complexity back into the model, one by one
- Priority-driven scheduling
    - Task-level fixed-priority (TFP): all jobs of periodic task have the same fixed priority
        - RM (rate-monotonic)
    - Task-level dynamic-priority: different priorities to individual jobs of a periodic task
        - Job-level fixed-priority (JFP): priority of each job is fixed
            - EDF (earliest deadline first)
        - Job-level dynamic-priority (JDP): priority of each job can change over time
            - LSTF (least slack time first)
- In analysis, what is the maximum utilization (rather than time)
    - Execution time / period

<u>10/13/21</u>

- Rate monotonic (RM)
    - A job is encountering worst-case (critical instant)
        - Shift each task so that its first job is released at t, just shifting the arrival times
    - If you can meet the job at the critical instant, then you can meet it in all cases
    - Utilization based analysis
        - Using CPU at 69.3%, you will be guaranteed to meet all jobs across a task set
        - U <= n(2^(1/n) - 1.0) -> 0.69
            - Calculate least upper bound of processor utilization
        - This is a sufficient condition, not a necessary condition
        - Example for 2 tasks
            - n = tasks, p = periods, t = tasks, e = execution times, U = utilization
            - if n=2, solve the equation and = 0.828
            - Let p2 < 2*p1
            - Determine the maximum schedulable e2
            - p2 <= p1 + e1, max(e2) = p1 - e1
            - p2 is in [p1, 2*p1]
        - Trying to have maximum job execution time without missing deadlines
        - Minimum U occurs when p2 = p1 + e1, where U = e1/p1 + (p1-e1)/(p1+e2)
            - Can take the derivative for p1 and set (partial derivative) dU/dp1 = 0
            - We get e1=(2^(½)-1.0)p1 and U=0.828
        - If total utilization is less than utilization upper bound function, then we're all good
            - Execution time / period = utilization (.753)

- .753 < .779
- Response-time analysis
  - $a_{n+1}=e_i+sum(a_n/p_j)e_j$
  - Test terminates when $a_{n+1} = a_n$
  - n tasks, testing schedulability of each task
    - Go down the line of tasks by priority and whether they're schedulable (assuming unlimited preemption)
  - a_n is estimation of response time or completion time (sums of higher priority tasks) of task i
  - Task i is schedulable if its response time is before its deadline:
    - $a_n <= p_i$
    - a_n is the response time of T_i
  - e1 = 40, e2 = 40, e3 = 100, p1 = 100, p2 = 150, p3 = 350
    - $a_0 = sum(e_j) = e_1+e_2+e_3 = 180$
    - $a_1 = 100 + 180/100(40) = 180/150(40) = 100 + 80 + 80 = 260$
    - $a_2 = e_i + sum(a_1/p_j)e_j = 100 + 260/100(40) + 260/150(40) = 300$
    - $a_3 = 100 + 300/100(40) + 300/150(40) = 300$
  - This works because time-demand analysis is based on the critical instant
    - If J_i is done at t, then the total work must be done in [0,t] is (from J_i and all higher priority tasks):
      - $w_i(t)=e_i+sum(t/p_k)e_k$

<u>10/15/21</u>

- Round robin
  - Similar to FCFS scheduling
  - CPU bursts (execution) assigned with time quantum
  - Advantages
    - Fairness equal share of CPU
    - New created process added to end of queue
    - Time sharing, each job/time slot has time quantum
    - Each process has a chance to reschedule
  - Disadvantage
    - Low throughput
    - Larger waiting time and response time
    - Context switches
    - Gantt chart becomes very big
    - Small quantums = time consuming
  - Metrics
    - Completion time
      - Time when process completes its execution
    - Turnaround time
      - Time difference between completion time and arrival time
    - Waiting time

- Time difference between turnaround time and burst time
- Rate-monotonic (RM)
  - Higher period frequency is higher priority task for RM
  - Response time analysis
    - Exact test, use if upper bound test is indeterminate
    - One analysis per task
  - Stop conditions
    - Deadline violation $R_{wci} > D_i = p_i$
    - Convergence $R_{wci}(m+1) = R_{wci}(m)$
- RM Example 1
  -

T_i	p_i	e_i
1	2	0.5
2	3	0.5
3	6	2

  - U = 0.5/2 + 0.5/3 + 2/6 = 0.75
  - U(3) = 0.779
  - 0.75 < 0.779
  - Sufficient, tasks are schedulable
- RM Example 2
  -

T_i	p_i	e_i
1	2	0.5
2	3	0.5
3	6	**3**

  - U = 0.92
  - U(3) = 0.779
  - 0.92 > 0.779
  - Is T_1 schedulable?
    - $R_{wc1}(0) = C_1 = 0.5 <= 2$
  - Is T_2 schedulable?
    - $R_{wc2}(0) = C_1 + C_2 = 1$
    - $R_{wc2}(1) = ceil(R_{wc2}(0)/T_1) * C_1 + C_2 = ceil(½) * 0.5 + 0.5 = 1$
    - Converged, 1<= 3
  - Is T_3 schedulable?
    - $R_{wc3}(0) = C_1 + C_2 + C_3 = 0.5 + 0.5 + 3 = 4$
    - $R_{wc3}(1) = ceil(R_{wc3}(0)/T_1) * C_1 + ceil(R_{wc3}(0)/T_2) * C_2 + C_3 =$
      - ceil(4/2) * 0.5 + ceil(4/3) * 0.5 + 3 = 5.5
    - $R_{wc3}(2) = ceil(R_{wc3}(1)/T_1) * C_1 + ceil(R_{wc3}(1)/T_2) * C_2 + C_3 =$

- 5.5
- Converged, 5.5 <= 6
- RM Example 3
  -

T_i	e_i
3	1
4	1
6	2.1

  - 0.93 > 0.779
  - $R_{wc3}(1) = ceil(R_{wc3}(0)/T_1) * C_1 + ceil(R_{wc3}(0)/T_2) * C_2 + C_3 = 6.1$
  - 6.1 > 6, deadline violation
- Number 6 don't try to derive the equation from the original RM equation

<u>10/20/21</u>

- RM Transient Overload
  - If task with lower period is not critical to the underlying application
  - To deal with this, consider period transformation, period aggregation, or period splitting
    - Could drop the task altogether, but this is non-desirable
    - Replace the problematic task with 2 tasks, each with 2x original period
- RM Schedulability With Interrupts
  - Interrupts should receive higher priority than application
  - Interrupt handler executes higher priority irrespective of its period
  - Interrupt processing can delay execution of app tasks with shorter periods
  - This interrupt processing must be accounted for in the schedulability model. How to change the UB test?
  - UB test with interrupt
    - Test is applied to each task
    - Determine effective utilization (f_i) of each task i using:
      - $Sum_{j=Hn}(e_i/p_i) + e_i/p_i + 1/p_i * Sum_{k=H1}(e_k)$
    - Compare effective utilization (f_i) to bound U(n)
      - n = num(H_n + 1) where num(H_n) = number of tasks in set H_n
      - H_n is the set of tasks that will preempt current task more than once with period less than D_i
      - H_1 is the set of tasks that preempt current task only once with period greater than D_i
- Priority inversion
  - Delay to a task's execution is when blocking occurs from lower-priority tasks
  - If the tasks share the same resources, this can happen
  - We need to identify and evaluate sources of priority inversion
  - Sources

        - Synchronization and mutual exclusion (mutex locks)
        - Non-preemptable regions of code
        - FIFO queues
    - How to deal with priority inversion in schedulability analysis
        - Task schedulability is affected by:
            - Preemption: 2 types
                - Occurs several times per task period OR
                - Occurs once per period
            - Execution: Once per period
            - Blocking: At most once per period for each resource
    - Schedulability formulas are modified to add a “blocking” or “priority inversion” term
    - Response time analysis with blocking
        - $a_{n+1} = B_i + e_i + Sum(a_n/p_i)e_i$
        - Perform test as done before, including blocking effect
            - Where $a_0 = B_i + Sum(e_j)$
    - Example
        - Where data structure is 30 msec to access
        - T_3 just enters the critical section, then T_2 preempts T_3 while T_1 is still waiting for the data structure, so T_1 must wait for T_2 to finish its computation
        -

Task	Period	Execution Time	Priority	Blocking Delay	Deadline
T_1	100	25	High	30+50	100
T_2	200	50	Medium	0	200
T_3	300	100	Low	0	300

        - $f_1 = e_1/p_1 + B_1/p_1 = 25/100 + 80/100 = 1.05$
            - 1.05 > 1.00, not schedulable
        - $f_2 = e_1/p_1 + e_2/p_2 = 0.5$
            - 0.5 < U(2)
        - $f_3 = e_1/p_1 + e_2/p_2 + e_3/p_3 = 0.84$
            - 0.84 > U(3)
        - Higher priority task is not always more schedulable than lower priority tasks because of this
- EDF schedulability analysis
    - EDF is schedulable iff U <= 1.0

<u>10/27/21 - Priority Inversion</u>

- Usually use mutex locks
- Priority inversion occurs with shared resources or critical section

- High priority task wants to access locked resource, but it has to wait since low priority task has locked it
  - Low priority task runs for a bit
  - Then medium priority task preempts it
  - Low priority task finishes and unlocks the resource
  - Finally high priority task can run
- Normally, priority inversion is not harmful
  - But it could cause serious problems
- Mars Pathfinder
  - Landed on Mars on July 4th 1997
  - Surface operations, daily images of Mars, daily weather reports from surface of Mars
  - On July 12th, there were technical problems (communication errors)
  - On July 19th, the problem was solved
  - Turns out, a CTO of the RTOS for Pathfinder said that there was a priority inversion problem
  - Pathfinder had 3 tasks:
    - T_H: information bus task (short, frequent, quick responses)
    - T_M: communication task (sending pictures to Earth)
    - T_L: meteorological task (long task)
    - T_H had to wait very long because of priority inversion
    - Usually you build a timeout mechanism to do a total system reset
- You can drive any system to an unknown state (digital upset) which causes a total system reset
  - Timeout based reset mechanism can be exploited
- Solution is a priority inheritance protocol or priority ceiling protocol