Final Review
PART I

Reliability
Reliability

- Reliability for a given mission duration $t$, $R(t)$, is the probability of the system working as specified (i.e., probability of no failures) for a duration that is at least as long as $t$.

- The most commonly used reliability function is the exponential reliability function:

$$ R(t) = e^{-\lambda t} $$

where $\lambda$ is the failure rate.

From queueing theory: Probability of zero independent arrivals in $t$ time units (Poisson arrival process)
Reliability

- The most commonly used reliability function is the exponential reliability function:

\[ R(t) = e^{-\lambda t} \]

where \( \lambda \) is the failure rate.

- Mean time to failure (MTTF): \( 1/\lambda \)
Simple Reliability Modeling

- Total failure rate \( \lambda_1 + \lambda_2 \)
- Mean time to failure \( = \frac{1}{\lambda_1 + \lambda_2} \)
- Total reliability:
  \[
  R(t) = r_1(t)r_2(t) = e^{-(\lambda_1 + \lambda_2)t}
  \]
Simple Reliability Modeling

Note: This system needs at least one of the two components to function.

- Total reliability:

\[ R(t) = 1 - (1 - r_1(t))(1 - r_2(t)) \]
Note: This system needs at least two of the three components to function.

Total reliability:

\[ R(t) = r^3(t) + 3r^2(t)(1 - r(t)) \]
Other Implications

\[ R(\text{Effort}, \text{Complexity}, t) = e^{-kC \frac{t}{E}} \]

- Note: splitting the effort greatly reduces reliability.
Simplex Architectural Pattern

A simple verifiable core; diversity in the form of 2 alternatives; feedback control of the software execution.

Better performance, but less reliable

Is safety guaranteed? If not, switch.
Well Formed Dependencies

- **Informal intuition:** A reliable component should not *depend* on a less reliable component (it defeats the purpose).

- **Design guideline:** *Use but do not depend* on less reliable components
Review of Important Theorems

- **Total Probability Theorem:**
  \[ P(A) = P(A|C_1) \cdot P(C_1) + \ldots + P(A|C_n) \cdot P(C_n) \]
  where \( C_1, \ldots, C_n \) cover the space of all possibilities

- **Bayes Theorem:**
  \[ P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \]

- **Other:**
  \[ P(A,B) = P(A|B) \cdot P(B) \]
PART I

Timeliness
Some Terminology

- Tasks, periods, arrival-time, deadline, execution time, etc.

![Diagram showing the relationship between time, arrival time, deadline, period, and execution time]

- Arrival time, $a_i$ (Release time, $r_i$)
- Execution time, $e_i$ (Computation time, $c_i$)
- Deadline, $d_i$
- Start time, $s_i$
- Finish time, $f_i$
- Relative Deadline, $D_i$
- Period, $P_i$
- Arrival of Next invocation
Utilization Bounds

$$\text{Utilization} = \sum_i \frac{C_i}{P_i}$$

- **Modified** Question: is there a threshold $U_{\text{bound}}$ such that
  - When $U < U_{\text{bound}}$ deadlines are met
  - When $U > U_{\text{bound}}$ deadlines **may or may not be** missed

$U < U_{\text{bound}}$ is a sufficient but not necessary schedulability condition

All green area (schedulable)
The Schedulability Condition

For \( n \) independent periodic tasks with periods equal to deadlines:

The utilization bound of EDF = 1.

The Utilization bound of RM is:

\[
U = n \left( 2^{\frac{1}{n}} - 1 \right)
\]

\( n \to \infty \quad U \to \ln 2 \)
Consider a system of two tasks:

Task 1: $P_1=1.7$, $D_1=0.5$, $C_1=0.5$
Task 2: $P_2=8$, $D_2=3.2$, $C_2=2$

$I = \sum_{j \in HP} \left[ \frac{R_i}{P_j} \right] C_j$

$I$: Interference of higher priority tasks, $HP$ with task $i$. 

$R_i = I + C_i$
Consider a system of two tasks:

Task 1: $P_1 = 1.7$, $D_1 = 0.5$, $C_1 = 0.5$

Task 2: $P_2 = 8$, $D_2 = 3.2$, $C_2 = 2$

$$I = \sum_{j \in HP} \left[ \frac{R_i}{P_j} \right] C_j$$

$$R_i = I + C_i$$

$I$: Interference of higher priority tasks, $HP$ with task $i$.

$I^{(0)} = C_1 = 0.5$

$R_2^{(0)} = I^{(0)} + C_2 = 2.5$

$I^{(1)} = \left[ \frac{R_2^{(0)}}{P_1} \right] C_1 = \left[ \frac{2.5}{1.7} \right] 0.5 = 1$

$R_2^{(1)} = I^{(1)} + C_2 = 3$

$I^{(2)} = \left[ \frac{R_2^{(1)}}{P_1} \right] C_1 = \left[ \frac{3}{1.7} \right] 0.5 = 1$

$R_2^{(2)} = I^{(2)} + C_2 = 3$

$3 < 3.2 \rightarrow \text{Ok!}$
Blocking and Priority Inversion

Consider the case below: a series of intermediate priority tasks is delaying a higher-priority one.

- Attempt to lock S results in blocking.

Diagram:
- High-priority task
- Intermediate-priority tasks
- Lock S
- Low-priority task
- Preempt.
- Unbounded Priority Inversion
Priority Inheritance Protocol

- Let a task inherit the priority of any higher-priority task it is blocking

High-priority task

Intermediate-priority tasks

Low-priority task

Attempt to lock S results in blocking

Unlock S

Unlock S

Preempt.
Maximum Blocking Time

- If all critical sections are equal (of length $B$):
  - Blocking time = $B \min(N, M)$
  (Why?)
- If they are not equal
  - Find the worst (maximum length) critical section for each resource
  - Add up the top $\min(N, M)$ sections in size
- The total priority inversion time for task $i$ is called $B_i$
Schedulability Test

\[ \forall i, 1 \leq i \leq n, \quad \frac{B_i}{P_i} + \sum_{k=1}^{i} \frac{C_k}{P_k} \leq i(2^{1/i} - 1) \]
Problem: Deadlock

Deadlock occurs if two tasks locked two semaphores in opposite order.

- Lock R1
- Try R1, Block
- Lock R2
- Try R2, Deadlock

Preemption
Priority Ceiling Protocol

- Definition: The priority ceiling of a semaphore is the highest priority of any task that can lock it.

- A task that requests a lock $R_k$ is denied if its priority is not higher than the highest priority ceiling of all currently locked semaphores (say it belongs to semaphore $R_h$).
  - The task is said to be blocked by the task holding lock $R_h$.

- A task inherits the priority of the top higher-priority task it is blocking.
Maximum Blocking Time

Priority Inheritance Protocol

Need Red
Need Blue
Need Yellow
Maximum Blocking Time

Priority Ceiling Protocol

Need Blue but Priority is lower Than Red ceiling

Need Yellow but Priority is lower Than Red ceiling

Need Red but Priority is lower Than Red ceiling

Done
Example of a Polling Server

- Polling server:
  - Period $P_s = 5$
  - Budget $B_s = 2$

- Periodic task
  - $P = 4$
  - $C = 1.5$

- All aperiodic arrivals have $C=1$
Deferrable Server

- Keeps the balance of the budget until the end of the period
- Example (continued)
Exercise: Derive the utilization bound for a deferrable server plus one periodic task
Priority Exchange Server

Example

Aperiodic tasks

Priority Exchange Server

Periodic Tasks
Sporadic Server

- Server is said to be **active** if it is in the running or ready queue, otherwise it is **idle**.
- When an aperiodic task comes and the budget is not zero, the server becomes active.
- Every time the server becomes active, say at $t_A$, it sets replenishment time one period into the future, $t_A + P_s$ (but does not decide on replenishment amount).
- When the server becomes idle, say at $t_I$, set replenishment amount to capacity consumed in $[t_A, t_I]$

\[
U_p \leq \ln\left(\frac{2}{U_s + 1}\right)
\]
Slack Stealing Server

- Compute a slack function $A(t_s, t_f)$ that says how much total slack is available
- Admit aperiodic tasks while slack is not exceeded
PART III

Energy
Power of Computation

- Terminology
  - $R$: Power spent on computation
  - $V$: Processor voltage
  - $f$: Processor clock frequency
  - $R_0$: Leakage power

- Power spent on computation is:
  - $R = k_v V^2f + R_0$
  - where $k_v$ is a constant
Energy of Computation

- Power spent on computation is:
  \[ R = k_v V^2 f + R_0 \]
- Consider a task of length \( C \) clock cycles and a processor operating at frequency \( f \)
- The execution time is \( t = C/f \)
- Energy spent is:
  \[ E = R \cdot t = (k_v V^2 f + R_0)(C/f) \]
Reducing Processor Frequency
Good or Bad?

Does it make sense to operate the processor at a reduced speed to save energy? Why or why not?

Possible Answer:

\[ E = R \cdot t = (k_v V^2 f + R_0)(C/f) = k_v V^2 C + R_0 C/f \]


- Conclusion: \( E \) is minimum when \( f \) is maximum.
  
  \[ \rightarrow \text{Operate at top speed} \]

- Is this really true? What are the underlying assumptions?
Dynamic Voltage Scaling (DVS):
Reducing Voltage and Frequency

- Processor voltage can be decreased if clock frequency is decreased:
  - Voltage and frequency can be decreased roughly proportionally.
  - In this case (where $V \sim f$):
    
    $$ R = k_f f^3 + R_0 $$
    
    $$ E = (k_f f^3 + R_0)(C/f) = k_f f^2 C + R_0 C/f $$
Dynamic Voltage Scaling (DVS):
Reducing Voltage and Frequency

- Processor voltage can be decreased if clock frequency is decreased
- Voltage and frequency can be decreased roughly proportionally.

\[ R = k_f f^3 + R_0 \]
\[ E = (k_f f^3 + R_0)(C/f) = k_f f^2 C + R_0 C/f \]

- Question: Does reducing frequency (and voltage) increase or decrease total energy spend on a task?
Dynamic Voltage Scaling (DVS): The Critical Frequency

- There exists a minimum frequency below which no energy savings are achieved.

\[ E = k_f f^2 C + R_0 C/f \]
\[ \frac{dE}{df} = 2k_f f C - R_0 C/f^2 = 0 \]

\[ f = 3 \sqrt[3]{\frac{R_0}{2k_f}} \]
DVS Algorithm 1: Static Voltage Scaling

1. Calculate the critical frequency
2. Calculate the minimum frequency at which the task set remains schedulable
   - Example: If EDF is used and the utilization is 60% at the maximum frequency $f_{max}$, then the frequency can be decreased to $0.6f_{max}$.
3. Let $f_{opt}$ be the larger of the above two
4. Operate the system at the smallest frequency at or above $f_{opt}$. 
DVS Algorithm 2: Cycle-conserving DVS

- What if a task finishes early?
  - Re-compute the utilization based on the reduced execution time.
  - Calculate the minimum frequency at which the task set is schedulable using the new utilization.
  - Update task execution times to the WCET when new invocations are released.
Practical Consideration: Accounting for Off-chip Overhead

- In the preceding discussion, we assumed that task execution time at frequency $f$ is $C/f$, where $C$ is the total cycles needed.

- In reality some cycles are lost waiting for memory access and I/O (Off-chip cycles).
  - Let the number of CPU cycles used be $C_{cpu}$ and the time spent off-chip be $C_{off-chip}$
  - Execution time at frequency $f$ is given by $C_{cpu}/f + C_{off-chip}$
Recap

DVS

Reduce Frequency Only
- Processor Sleeps when Idle
  - Bad idea!
- Processor Always On
  - Good idea!

Reduce Frequency and Voltage
- Processor Sleeps when Idle
  - Good idea down to a Critical Frequency only
Processor Performance States (P-States)

- **P0** max power and frequency
- **P1** less than P0, voltage/frequency scaled
- **P2** less than P1, voltage/frequency scaled
- ...
- **Pn** less than $P(n-1)$, voltage/frequency scaled
Processor “Sleep” States (C-states)

- **C0**: is the operating state.
- **C1** (often known as *Halt*): is a state where the processor is not executing instructions, but can return to an executing state instantaneously. All ACPI-conformant processors must support this power state.
- **C2** (often known as *Stop-Clock*): is a state where the processor maintains all software-visible state, but may take longer to wake up. This processor state is optional.
- **C3** (often known as *Sleep*): is a state where the processor does not need to keep its cache, but maintains other state. This processor state is optional.
Turning Processors Off
The Cost of Wakeup

- Energy expended on wakeup, $E_{\text{wake}}$
- To sleep or not to sleep?
  - Not to sleep (for time $t$):
    \[ E_{\text{no-sleep}} = (k_v V^2 f + R_0) t \]
  - To sleep (for time $t$) then wake up:
    \[ E_{\text{sleep}} = P_{\text{sleep}} t + E_{\text{wake}} \]
  - To save energy by sleeping: $E_{\text{sleep}} < E_{\text{no-sleep}}$

\[ t > \frac{E_{\text{wake}}}{k_v V^2 f + R_0 - P_{\text{sleep}}} \]
DPM and the Problem with Work-conserving Scheduling

- No opportunity to sleep 😞

Task 1 (C=2, P=12)

Task 2 (C=1, P=16)

Minimum sleep period
DPM and the Problem with Work-conserving Scheduling

- Must batch! 😊

Task 1 (C=2, P=12)

Task 2 (C=1, P=16)

Minimum sleep period
How Many Processors to Use?

- Consider using one processor at frequency $f$ versus two at frequency $f/2$

- Case 1: Total power for one processor
  - $k_f f^3 + R_0$

- Case 2: Total power for two processors
  - $2 \{ k_f (f/2)^3 + R_0 \} = k_f f^3/4 + 2 R_0$

- The general case: $n$ processors
  - $n \{ k_f (f/n)^3 + R_0 \} = k_f f^3/ n^2 + n R_0$
How Many Processors to Use?

The general case: $n$ processors

$\begin{align*}
\text{Power} &= n \left\{ k_f \left( f/n \right)^3 + R_0 \right\} = k_f f^3/n^2 + n R_0 \\
d\text{Power}/dn &= -2 k_f f^3/n^3 + R_0 = 0
\end{align*}$

$$n = 3 \sqrt{\frac{2 k_f f^3}{R_0}}$$
Classical Feedback Control Loops

- Desired Set Point
- Measured Output
- Output
- Controller (Policy)
- Actuator (Mechanism)
- Process
- Sensor
- Feedforward
Stability – Recap

Phase equation: \( \sum_i p_i(f) = \pi \) \( \rightarrow \) \( f \) is obtained

Gain equation: \( \prod_i g_i(f) \) must be less than 1 for stability

\[
\begin{align*}
T_r & \quad e \\ 
& \quad - \\
& \quad g_1(f) / p_1(f) \\
& \quad g_2(f) / p_2(f) \\
& \quad g_3(f) / p_3(f) \\
& \quad u \\
& \quad m \\
& \quad T_m \\
& \quad T_m \\
& \quad T \\
\end{align*}
\]
## Summary of Basic Elements

Input = \sin (wt)

<table>
<thead>
<tr>
<th>Element</th>
<th>Gain</th>
<th>Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrator</td>
<td>\frac{1}{w}</td>
<td>-\pi/2</td>
</tr>
<tr>
<td>Differentiator</td>
<td>\omega</td>
<td>\pi/2</td>
</tr>
<tr>
<td>Pure delay element</td>
<td>1</td>
<td>-wD</td>
</tr>
<tr>
<td>(Delay = D)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>First order lag (time constant = \tau)</td>
<td>\frac{K}{\sqrt{1+(\tau\omega)^2}}</td>
<td>-\tan^{-1} (w \tau)</td>
</tr>
<tr>
<td>Pure gain (Gain = K)</td>
<td>K</td>
<td>0</td>
</tr>
</tbody>
</table>

Note:

\[ w = 2 \pi f_{osc} \]

Where \( f_{osc} \) is the loop frequency of oscillation
Steady State Error

At steady state the system “catches up” – phase shift is zero.

\[ T_r - e \cdot g_1(f) \cdot g_2(f) \cdot g_3(f) \cdot g_4(f) = e \]

\[ e \approx \frac{T_r}{1 + \prod_i g_i(f)} \]