Linux Tutorial

- 2/17 (Thursday)

- Lecture by Ruixuan (Corey) Dai

- Use a different Zoom link
Demo I

- **In person** on 3/1 and 3/3
- **10 min** per team
  - 9-min demo + 1-min Q&A
- Must show something **real**
- Submit slides before the class of your demo
Critique #1

Due: 10 am, March 8 (Tuesday)

Paper

- Available on the reading list on the class website.

Follow Critique Guidelines on the class website
Real-Time Scheduling

*Single CPU*

Chenyang Lu
Real-Time Scheduling

- What are the optimal scheduling algorithms?
- How to assign priorities to tasks?
- Can a system meet all deadlines?
## Benefit of Scheduling Analysis

- Schedulability analysis reduces development time by **50%**!
  - Reduce wasted implementation/testing rounds
  - Analysis time \(<<\) testing
- Quick exploration of design space!
  - More reduction expected for more complex systems

<table>
<thead>
<tr>
<th></th>
<th>VEST (UVA)</th>
<th>Baseline (Boeing)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design – one processor</td>
<td>40</td>
<td>Design – one processor</td>
</tr>
<tr>
<td>Implementation – one processor</td>
<td></td>
<td>Implementation – one processor</td>
</tr>
<tr>
<td>Scheduling analysis - MUF</td>
<td>1</td>
<td>Timing test</td>
</tr>
<tr>
<td>Design - two processors</td>
<td>25</td>
<td>Design - two processors</td>
</tr>
<tr>
<td>Implementation – two processors</td>
<td></td>
<td>Timing test</td>
</tr>
<tr>
<td>Scheduling analysis - DM/Offset</td>
<td>1</td>
<td>Timing test</td>
</tr>
<tr>
<td>“Implementation”</td>
<td>105</td>
<td>Total composition time</td>
</tr>
<tr>
<td>Total composition time</td>
<td>172</td>
<td></td>
</tr>
</tbody>
</table>

Terminology

- **Task**
  - Map to a process or thread
  - May be released multiple times
  - **Job**: an instance of a task

- **Periodic task**
  - Ideal: inter-arrival time = period
  - General: inter-arrival time $\geq$ period

- **Aperiodic task**
  - Inter-arrival time does not have a lower bound
Timing Parameters

- Task $T_i$
  - Period $P_i$, worst-case execution time $C_i$, relative deadline $D_i$

- Job $J_{ik}$
  - Release time: time when a job is ready
  - Response time $R_i = \text{finish time} - \text{release time}$
  - Absolute deadline = release time + $D_i$

- A job misses its deadline if
  - Response time $R_i > D_i$
  - Finish time > absolute deadline

\[\begin{align*}
P_1 &= D_1 = 5, \quad C_1 = 2; \\
P_2 &= D_2 = 7, \quad C_2 = 4.
\end{align*}\]
Metrics

- A task set is **schedulable** if all jobs meet their deadlines.

- **Optimal** scheduling algorithm
  - A task set is unschedulable under the optimal algorithm \( \rightarrow \)
  unschedulable under any other algorithms.

- **Overhead**: Time required for scheduling
Optimal Scheduling Algorithms

- **Rate Monotonic (RM)**
  - Higher rate (1/period) $\rightarrow$ Higher priority
  - Optimal preemptive *static* priority scheduling algorithm

- **Earliest Deadline First (EDF)**
  - Earlier *absolute* deadline $\rightarrow$ Higher priority
  - Optimal preemptive *dynamic* priority scheduling algorithm
Example

- $P_1 = D_1 = 5, C_1 = 2; P_2 = D_2 = 7, C_2 = 4.$

Chenyang Lu
Assumptions

- Single CPU.
- All tasks are periodic.
- Zero context switch time.
- Relative deadline = period.
- No priority inversion.

- Have been extended to remove these assumptions.
Schedulable Utilization Bound

• **Utilization of a CPU:**

\[ U = \sum_{i=1}^{n} \frac{C_i}{P_i} \]

– \( n \): number of tasks

• **Utilization bound \( U_b \)**
  – All tasks are guaranteed to be schedulable if \( U \leq U_b \).

• **No scheduling algorithm can schedule a task set if \( U > 1 \)**
  – \( U_b \leq 1 \)
  – An algorithm is optimal if its \( U_b = 1 \)
RM Utilization Bound

- \( U_b(n) = n(2^{1/n} - 1) \)
  - \( n \): number of tasks
  - \( U_b(2) = 0.828 \)
  - \( U_b(n) \geq U_b(\infty) = \ln 2 = 0.693 \)

- \( U \leq U_b(n) \) is a sufficient condition, but not necessary.

- \( U_b = 1 \) if all task periods are harmonic
  - Periods are multiples of each other
  - e.g., 1, 10, 100
Properties of RM

- May not guarantee schedulability when CPU is not fully utilized.

- Low overhead
  - When the task set is fixed, the priority of a task never changes.

- Easy to implement on POSIX APIs.
EDF Utilization Bound

- $U_b = 1$
  - $U \leq 1$: sufficient and necessary condition for schedulability.

- Guarantees schedulability if CPU is not over-utilized.

- Higher overhead than RM: task priority may change online.
Assumptions

- Single processor.
- All tasks are periodic.
- Zero context switch time.
- Relative deadline = period.
- No priority inversion.

- What if relative deadline < period?
Optimal Scheduling Algorithms

Relative Deadline < Period

- Deadline Monotonic (DM)
  - Shorter relative deadline → Higher priority
  - Optimal preemptive static priority scheduling

- Earliest Deadline First (EDF)
  - Earlier absolute deadline → Higher priority
  - Optimal preemptive dynamic priority scheduling algorithm
DM Analysis

• Sufficient but pessimistic test

\[ \sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{1/n} - 1) \]

• Sufficient and necessary test: response time analysis
Response Time Analysis

• Works for any fixed-priority preemptive scheduling algorithm.
• Critical instant
  – results in a task’s longest response time.
  – when all higher-priority tasks are released at the same time.
• Worst-case response time
  – Tasks are ordered by priority; $T_1$ has highest priority

$$R_i = C_i + \sum_{j=1}^{i-1} \left[ \frac{R_i}{T_j} \right] C_j$$

*Number of jobs of $T_j$ to preempt a job of $T_i$*
Interference of Higher-priority Tasks

<table>
<thead>
<tr>
<th>$\tau_1$</th>
<th>$\tau_2$</th>
<th>$\tau_3$</th>
<th>$\tau_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_i$</td>
<td>$T_i$</td>
<td>$D_i$</td>
<td>$C_i$</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
DM_guarantee (Γ) {
    for (each τ_i ∈ Γ) {
        I_i = \sum_{k=1}^{i-1} C_k;
        do {
            R_i = I_i + C_i;
            if (R_i > D_i) return(UNSCHEDULABLE);
            I_i = \sum_{k=1}^{i-1} \left\lfloor \frac{R_i}{T_k} \right\rfloor C_k;
        } while (I_i + C_i > R_i);
    }
    return(SCHEDULABLE);
}
Example

<table>
<thead>
<tr>
<th></th>
<th>$C_i$</th>
<th>$T_i$</th>
<th>$D_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>1</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

Step 0: $R_4^{(0)} = \sum_{i=1}^{4} C_i = 5$, but $I_4^{(0)} = 5$ and $I_4^{(0)} + C_4 > R_4^{(0)}$ hence $\tau_4$ does not finish at $R_4^{(0)}$.

Step 1: $R_4^{(1)} = I_4^{(0)} + C_4 = 6$, but $I_4^{(1)} = 6$ and $I_4^{(1)} + C_4 > R_4^{(1)}$ hence $\tau_4$ does not finish at $R_4^{(1)}$.

Step 2: $R_4^{(2)} = I_4^{(1)} + C_4 = 7$, but $I_4^{(2)} = 8$ and $I_4^{(2)} + C_4 > R_4^{(2)}$ hence $\tau_4$ does not finish at $R_4^{(2)}$.

Step 3: $R_4^{(3)} = I_4^{(2)} + C_4 = 9$, but $I_4^{(3)} = 9$ and $I_4^{(3)} + C_4 > R_4^{(3)}$ hence $\tau_4$ does not finish at $R_4^{(3)}$.

Step 4: $R_4^{(4)} = I_4^{(3)} + C_4 = 10$, but $I_4^{(4)} = 9$ and $I_4^{(4)} + C_4 = R_4^{(4)}$ hence $\tau_4$ finishes at $R_4 = R_4^{(4)} = 10$. 

Chenyang Lu
EDF: Processor Demand Analysis

• To start, assume $D_i = P_i$

• Processor demand in interval $[0, L]$: total time needed for completing all jobs with deadlines no later than $L$.

$$C_P(0, L) = \sum_{i=1}^{n} \left\lfloor \frac{L}{P_i} \right\rfloor C_i$$
A set of periodic tasks is schedulable by EDF if and only if for all $L \geq 0$:

$$L \geq \sum_{i=1}^{n} \left\lceil \frac{L}{P_i} \right\rceil C_i$$

There is enough time to meet processor demand at every time instant.
Busy Period $B_p$

- End at the first time instant $L$ when all the released jobs are completed.
- $W(L)$: Total execution time of all tasks released by $L$.

$$W(L) = \sum_{i=1}^{n} \left\lfloor \frac{L}{P_i} \right\rfloor C_i$$

$$B_p = \min\{L \mid W(L) = L\}$$
Properties of Busy Period

- CPU is fully utilized during a busy period.
- The end of a busy period coincides with the beginning of an idle time or the release of a periodic job.
All tasks are schedulable if and only if

\[ L \geq \sum_{i=1}^{n} \left\lfloor \frac{L}{P_i} \right\rfloor C_i \]

at all job release times before \( \min(B_p, H) \)
Compute Busy Period

```c
busy_period
{
    H = lcm(P_1, ..., P_n); /* least common multiple */
    L = \sum C_i;
    L' = W(L);
    while (L' != L and L' <= H) {
        L = L';
        L' = W(L);
    }
    if (L' <= H)
        B_p = L;
    else
        B_p = INFINITY;
}
```
Processor Demand Test: $D_i < P_i$

- A set of periodic tasks with deadlines no more than periods is schedulable by EDF if and only if

$$\forall L \in D, \quad L \geq \sum_{i=1}^{n} \left( \left\lceil \frac{L - D_i}{P_i} \right\rceil + 1 \right) C_i$$

where $D = \{D_{i,k} \mid D_{i,k} = kP_i + D_i, D_{i,k} \leq \min(B_p, H), 1 \leq i \leq n, k \geq 0\}$.

- Note: only need to test all deadlines before $\min(B_p, H)$.
## Schedulability Test Revisited

<table>
<thead>
<tr>
<th></th>
<th>( D = P )</th>
<th>( D &lt; P )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static Priority</strong></td>
<td>RM Utilization bound Response time</td>
<td>DM Response time</td>
</tr>
<tr>
<td><strong>Dynamic Priority</strong></td>
<td>EDF Utilization bound</td>
<td>EDF Processor demand</td>
</tr>
</tbody>
</table>

Check out examples at [http://www.cse.wustl.edu/~lu/cse467s/slides/example_sched.pdf](http://www.cse.wustl.edu/~lu/cse467s/slides/example_sched.pdf)
Assumptions

- Single processor.
- All tasks are periodic.
- Zero context switch time.
- Relative deadline = period.
- No priority inversion.
Questions

- What causes priority inversion?
- How to reduce priority inversion?
- How to analyze schedulability?
Priority Inversion

- A low-priority task blocks a high-priority task.

- Sources of priority inversion
  - Access shared resources guarded by semaphores.
  - Access non-preemptive subsystems, e.g., storage, networks.
Semaphores

- OS primitive for controlling access to critical regions.
  - Get access to semaphore S with `sem_wait(S)`.
  - Perform critical region operations.
  - Release semaphore with `sem_post(S)`.

- Mutex: only one process can hold a mutex at a time.

```
sem_wait(mutex_info_bus);
Write data to info bus;
sem_post(mutex_info_bus);
```
What happened to Pathfinder?

…But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data…

Real-World (Out of This World) Story: Priority inversion almost ruined the path finder mission on MARS! [http://research.microsoft.com/~mbj/](http://research.microsoft.com/~mbj/)
Priority Inversion

T₁ blocked!

critical section
Unbounded Priority Inversion

T₁ blocked by T₄, T₂, T₃!

Critical section
The low-priority task **inherits** the priority of the blocked high-priority task.
Priority Inheritance Protocol (PIP)

- When task $T_i$ is blocked on a semaphore held by $T_k$
  - If $\text{prio}(T_k)$ is lower than $\text{prio}(T_i)$, $\text{prio}(T_i) \rightarrow T_k$

- When $T_k$ releases a semaphore
  - If $T_k$ no longer blocks any tasks, it returns to its normal priority.
  - If $T_k$ still blocks other tasks, it inherits the highest priority of the remaining tasks that it is blocking.

- Priority Inheritance is transitive
  - $T_2$ blocks $T_1$ and inherits $\text{prio}(T_1)$
  - $T_3$ blocks $T_2$ and inherits $\text{prio}(T_1)$
How was Path Finder saved?

- When created, a VxWorks mutex object accepts a boolean parameter that indicates if priority inheritance should be performed by the mutex.
  - The mutex in question had been initialized with the parameter **FALSE**.

- VxWorks contains a C interpreter intended to allow developers to type in C expressions/functions to be executed on the fly during system debugging.

- The initialization parameter for the mutex was stored in global variables, whose addresses were in symbol tables also included in the launch software, and available to the C interpreter.

- A C program was uploaded to the spacecraft, which when interpreted, changed these variables from **FALSE** to **TRUE**.

- **No more system resets occurred.**

---

Bounded Number of Blocking

- Assumptions of analysis
  - Fixed priority scheduling
  - All semaphores are binary
  - All critical sections are properly nested

- Task $T_i$ can be blocked by at most $\min(m, n)$ times
  - $m$: number of distinct semaphores that can be used to block $T_i$
  - $n$: number of lower-priority tasks that can block $T_i$
Extended RMS Utilization Bound

• A set of periodic tasks can be scheduled by RMS/PIP if

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

- Tasks are ordered by priorities (T_1 has the highest priority).
- B_i: the maximum amount of time when task T_i can be blocked by a lower-priority task.
Consider the effect of blocking on response time:

\[ R_i = C_i + B_i + \sum_{j=1}^{i-1} \left( \frac{R_i}{P_j} \right) C_j \]

The analysis becomes sufficient but not necessary.
Priority Ceiling

- $C(S_k)$: Priority ceiling of a semaphore $S_k$
  - Highest priority among tasks requesting $S_k$.

- A critical section guarded by $S_k$ may block task $T_i$ only if $C(S_k)$ is higher than $\text{prio}(T_i)$
Compute $B_i$

Assumption: no nested critical sections.

/* potential blocking by other tasks */
B1=0; B2=0;
for each $T_j$ with priority lower than $T_i$ {
    $b_1 =$ longest critical section in $T_j$ that can block $T_i$
    B1 = B1 + b1
}

/* potential blocking by semaphores */
for each semaphore $S_k$ that can block $T_i$ {
    $b_2 =$ longest critical section guarded by $S_k$ among lower priority tasks
    B2 = B2 + b2
}
return min(B1, B2)
Priority Ceiling Protocol

- **Priority ceiling of the CPU**: The highest priority ceiling of all semaphores currently held.

- A task can acquire a resource only if
  - the resource is free, AND
  - it has a higher priority than the priority ceiling of the CPU.

- A task can be blocked by at most **one** critical section.

- Higher run-time overhead than PIP.
Assumptions

- Single processor.
- All tasks are periodic.
- Zero context switch time.
- Relative deadline = period.
- No priority inversion.
Hybrid Task Set

- Periodic tasks + aperiodic tasks

- Problem: arrival times of aperiodic tasks are unknown

- Sporadic task with a hard deadline
  - Inter-arrival time must be lower bounded
  - Schedulability analysis: treated as a periodic task with period = minimum inter-arrival time → can be very pessimistic.

- Aperiodic task with a soft deadline
  - Possibly unbounded inter-arrival time
  - Maintain hard guarantees on periodic tasks
  - Reduce response time of aperiodic tasks
Background Scheduling

- Assign aperiodic tasks the lowest priority

- Advantages
  - Simple
  - Aperiodic tasks usually have no impact on periodic tasks.

- Disadvantage
  - Aperiodic tasks can have very long response times when the utilization of periodic tasks is high.

- Acceptable only if
  - System is not busy
  - Aperiodic tasks can tolerate long delays
Polling Server

- A periodic task serving aperiodic requests
  - Period: $P_s$
  - Capacity: $C_s$

- Released periodically at period $P_s$

- Serves any pending aperiodic requests

-Suspends itself till the end of the period when
  - it has used up its capacity, or
  - no aperiodic request is pending

- Capacity is replenished to $C_s$ at the beginning of the next period
Example: Polling Server

<table>
<thead>
<tr>
<th></th>
<th>$C_i$</th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Server

$C_s = 2$
$T_s = 5$

Figure 5.3  Example of a Polling Server scheduled by RM.
A polling server has the same impact on periodic tasks as a periodic task.

- n tasks with m servers: \( U_p + U_s \leq U_b(n+m) \)

Disadvantage: If an aperiodic request “misses” the server, it has to wait till the next period \( \rightarrow \) long response time.

Can have multiple servers (with different periods) for different classes of aperiodic requests
Deferrable Server (DS)

- Preserve unused capacity till the end of the current period → shorter response time for aperiodic requests.

- Impact on periodic tasks differs from a periodic task.
Example: Deferrable Server

<table>
<thead>
<tr>
<th>$\tau_1$</th>
<th>$C_i$</th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Server

$C_s = 2$

$T_s = 5$

Figure 5.7 Example of a Deferrable Server scheduled by RM.
RM Utilization Bound with DS

• Under RMS

\[ U_b = U_s + n \left[ \left( \frac{U_s + 2}{2U_s + 1} \right)^{1/n} - 1 \right] \]

• As \( n \rightarrow \infty \):

\[ U_b = U_s + \ln \left( \frac{U_s + 2}{2U_s + 1} \right) \]

– When \( U_s = 0.186 \), \( \min U_b = 0.652 \)

• System is schedulable if

\[ U_p \leq \ln \left( \frac{U_s + 2}{2U_s + 1} \right) \]
First DS implementation on top of priority-based OS (e.g., Linux, POSIX)
Server thread processes aperiodic events (2nd highest priority)
Budget manager thread (highest priority) manages the budget and controls the execution of server thread

Assumptions

- Single processor.
- All tasks are periodic.
- Zero context switch time.
- Relative deadline = period.
- No priority inversion.
Context Switch Time

- **RTOS usually has low context switch overhead.**

- Context switches can still cause overruns in a tight schedule.
  - Leave margin in your schedule.

- Techniques exist to reduce the number of context switches by avoiding certain preemptions.

- Other forms of overhead: cache, thread migration, interrupt handling, bus contention, thread synchronization…
Fix an Unschedulable System

- Reduce task execution times
- Reduce blocking factors
- Get a faster processor
- Multi-core, multi-processor, and distributed systems
- Replace software components with hardware
Readings

Hard Real-Time Computing Systems, by G. Buttazzo
- Chapter 4 Periodic Task Scheduling
- Chapter 5 (5.1-5.4) Fixed Priority Servers
- Chapter 7 (7.1-7.3) Resource Access Protocols

Further references