Overview

- Processes/Tasks and Concurrency
- Scheduling Priorities and Policies
- Multitasking
- Real-time Scheduling
  - Fixed-Priority and Earliest Deadline First Scheduling
- Sporadic and Aperiodic Process Scheduling

Chapter 6 of the Text by Wayne Wolf, Chapter 13 of Text by Burns and Wellings
Introduction to Processes

All multiprogramming operating systems are built around the concept of processes.

Process is also called a task.

OS and Processes

• OS must interleave the execution of several processes to maximize CPU usage.

  Keeping reasonable/minimum response time

• OS must allocate resources to processes.

  By avoiding deadlock

• OS must also support:
  ▪ IPC: Inter-process communication
  ▪ Creation of processes by other processes
Task/Process Concept

Serial Execution of Two Processes

Interleaving the Execution of Process 1 and 2
Processes and Managing Timing Complexity

Multiple rates
- multimedia
- automotive

Asynchronous Input
- user interfaces
- communication systems

Engine Control Tasks
- spark control
- crankshaft sensing
- fuel/air mixture
- oxygen sensor
- Kalman filter
Concurrency

• Only one thread runs at a time while others are \textit{waiting}.
• Processor switches from one process to another so quickly that it appears all threads are running simultaneously. Processes run \textit{concurrently}.
• Programmer assigns \textit{priority} to each process and the \textit{scheduler} uses it to determine which process to run next.

Real-Time Kernel

• Processes call a library of run-time routines (known as the real-time \textit{kernel}) manages resources.
• Kernel provides mechanisms to switch between processes, for coordination, synchronization, communications, and priority.
Process Context

- Each process has its own stack and context.
- A context switch from process "A" to process "B" first saves registers in context A, and then reloads all CPU registers from context B.
Basic Process States

There are three basic states of a process

- The Running state
  - The process that gets executed. (Max of one for one CPU)
- The Ready state
  - A process is ready to be executed.
- The Blocked state (Waiting)
  - When a process cannot execute until some event occurs.
    (e.g. completion of an I/O)

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
More Process States

5-State Process Model

- **New**: Admit
  - Commitment to execution when ready (in terms of # of processes & memory)

- **Ready**: Event Occurs
  - Scheduling-allowed

- **Running**: Event Wait
  - File request
  - I/O
  - IPC
  - Time-out or interrupt

- **Blocked**: Not eligible for Execution but Info preserved temporarily

- **Exit**: Release
  - At parent request
Process Transitions

- **Ready ⇒ Running**
  - Dispatcher selects a new process to run.
    - When the turn comes.
- **Running ⇒ Ready**
  - Running process has expired its time slot.
  - A higher priority process is in the ready state.
- **Running ⇒ Blocked** *(waiting)*
  - When a process requests something for which it must wait.
    - A service that the OS is not ready to perform.
    - An access to a resource not yet available.
    - Initiates I/O and must wait for the result.
    - Waiting for a process to provide input (IPC).
- **Blocked ⇒ Ready**
  - When the event, for which process is waiting, occurs.
Process Modes of Execution

Most processors support at least two execution modes:

- **Privileged mode**
  - System mode, kernel mode, supervisor mode,
    - Manipulating control registers
    - Memory management ...

- **User mode**
  - Less-privileged mode
  - User programs execute in this mode.

Therefore CPU provides a (or a few) mode bit, which may only be set by an interrupt or trap or OS call.
Process Hierarchies

- Forms a hierarchy
  UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  All processes are created equally.

- Process A created two child processes, B and C.
- Process B created three child processes, D, E & F.
UNIX Processes

- 2 modes: User mode and Kernel mode.
- System processes run in Kernel mode.
- User processes run in user mode for user instructions and in kernel mode for OS/kernel instructions.
- 9 states for processes.

UNIX Process State

- Two running states for user or kernel modes.
- Pre-empted state is for processes returning from Kernel to user mode.
  
  Kernel schedules another higher-priority process.

- A process running in Kernel mode cannot be pre-empted. This makes UNIX unsuitable for real-time. More later.
UNIX Process Transition Diagram

Two running states: User and Kernel
Preempted State: Kernel schedules another high priority process.
A Process running in Kernel mode cannot be preempted. That makes Unix/Linux unsuitable for real-time applications
UNIX Process Creation

Every process, except process 0, is created by the fork() system call.

- `fork()` allocates entry in process table and assigns a unique PID to the child process
- Child gets a copy of process image of parent: both child and parent are executing the same code following fork().
- `fork()` returns the PID of the child to the parent process and returns 0 to the child process.

Process 0 is created at boot time and becomes the “swapper” after forking process 1 (the INIT process)

When a user logs in: process 1 creates a process for that user.
UNIX-style Process Creation

int fork()
  - Creates an exact copy of the calling process.

int execve(char *progName, char *argv[ ])
  - Runs a new program in the calling process
  - Destroying the old program

int exit(int retCode)
  - Exits the calling process

int wait(int *retCode)
  - Waits for any exited child, returns its pid

<table>
<thead>
<tr>
<th>Process management</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Call</strong></td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>pid = fork( )</td>
</tr>
<tr>
<td>pid = waitpid(pid, &amp;statloc, options)</td>
</tr>
<tr>
<td>s = execve(name, argv, environp)</td>
</tr>
<tr>
<td>exit(status)</td>
</tr>
</tbody>
</table>
UNIX Fork

Processes in memory

Process 176

Process pointer ->

Copy

Process 284

Process 176’s memory

An exact and complete copy

7620: sys fork

7622:

Program stack

Return value = 284

7620: sys fork

7622:

Program stack

Return value = 0

Everything is identical except the value returned by the system call
#include <sys/types.h>
#include <stdio.h> #include <unistd.h>

int main()
{
    pid_t pid;
    pid = getpid();   /* Parent process created, get its ID */
    pid = fork();    /* Create a child process */
    if (pid == 0)
    {
        /* only the child process code should get here */
        while(1) {
            fprintf(stderr, "I am child process \n");
            usleep(10000000); /* wait for 10 seconds */
        }
    }
    /* Only parent should get here */
    fprintf(stderr," I am PARENT: I wait for 20 seconds\n");
    usleep(20000000);
    fprintf(stderr,"I am PARENT: Kill child: %u\n",pid);
    kill(pid,9);
    return(0);
}
Process Switching

A process switch may occur whenever the OS gain control of the CPU.

• Supervisor Call
  φ Transfer control to a piece of OS code (e.g. file open).
  φ Process may be switched to a blocked state.

• Trap
  φ An error resulted from the last instruction.
    Process moves to Exit state.

• Interrupt by an external independent event.
  φ Clock Interrupt: process has executed for the maximum allowable time slice. Switch to Blocked state.
  φ I/O Interrupt: OS moves waiting processes to READY
  φ Memory Fault: Memory address block is not in virtual memory so it must be brought into main memory.
    Move process to blocked state. (Waiting for the I/O to complete)
Process/Task Switching

How to change a process state

- Save context of processor including PC and other registers
- Update the PCB/TCB (process/task control block) with the new state and other associated information. e.g. accounting
- Move PCB to appropriate queue. Ready, blocked, suspend.
- Select another process for execution. Scheduling decision
- Update the process (task) control block of the process (task) selected.
- Update memory-management data structures
- Restore context of the selected process by reloading previous PC and registers.
Foreground/Background Multitasking System

Start

Initialize

Wait for Interrupts

Interrupt

ISR for Task #1

IRET

Interrupt

ISR for Task #2

IRET

Interrupt

ISR for Task #3

IRET
Foreground/Background System

• Most of the actual work is performed in the "foreground" ISRs, with each ISR processing a particular hardware event.
• Main program performs initialization and then enters a "background" loop that waits for interrupts to occur.
• System responds to external events with a predictable amount of latency.

Moving to Background
• Move non-time-critical work (such as updating a display) into background task.
• Foreground ISR writes data to queue, then background removes and processes it.
• An alternative to ignoring one or more interrupts as the result of input overrun.
Limitations of the Foreground/Background Multitasking

- Best possible performance requires moving as much as possible into the background.
- Background becomes collection of queues and associated routines to process the data.
- Optimizes latency of the individual ISRs, but background requires a managed allocation of processor time.
Co-operative Multitasking

- Hides context switching mechanism;
- Still relies on processes to give up CPU.
- Each process allows a context switch at cswitch() call.
- Separate scheduler chooses which process runs next.

Context switching
Who controls when the context is switched?
How is the context switched?

Problems with co-operative multitasking

Programming errors can keep other processes out:
- Process never gives up CPU;
- Process waits too long to switch, missing input.
Context Switching

Must copy all registers to activation record, keeping proper return value for PC.
Must copy new activation record into CPU state.
How does the program that copies the context keep its own context?

**Context switching in ARM**

**Start new process:**

```
ADR r0,NEXTPROC
LDR r13,[r0]
LDMDB r13,{r0,r14}
MSR SPSR,r0
LDMIA r13,{r0-r14}
MOV pc,r14
```

**Save old process:**

```
STMIA r13,{r0-r14}
MRS r0,SPSR
STMDB r13,{r0,r15}
```
Preemptive Multitasking

- Most powerful form of multitasking
- OS controls when contexts switches
- OS determines what process runs next
- Use timer to call OS, switch contexts:

Flow of control with preemption:

```
interrupt
CPU
interrupt
```

```
P1  OS  P1  OS  P2
interrupt
```
Preemptive Context Switching

- Timer-interrupt gives control to OS, which saves interrupted process’s state in an activation record.
- OS chooses next process to run.
- OS installs desired activation record as current CPU state.

**Why not use interrupts**

We could change the interrupt vector at every period, but:
- We would need management code anyway;
- We would have to know the next period’s process at the start of the current process.
Non-Preemptive Context Switch

State of Process A:
- Running
- Interrupted
- Running
- Ready

Interrupt Routine

Run-Time Kernel

State of Process B:
- Blocked
- Ready
- Running

Context Switch
Non-Preemptive Context Switch
Preemptive Context Switch

State of Process A
- Running
- Interrupted
- Ready

Interrupt Routine

State of Process B
- Blocked
- Ready
- Interrupted
- Running

Run-Time Kernel

Context Switch
Preemptive Context Switch

- Pending
- Running
- Ready
- Inactive
- Interrupted
VxWorks Multitasking

Modern real-time systems are based on the complementary concepts of multitasking and inter-task communications.

In VxWorks, tasks have immediate, shared access to most system resources, while also maintaining separate context to maintain individual task control.

A multitasking environment allows a real-time application to be constructed as a set of independent tasks, each with its own thread of execution and set of system resources.

It is often essential to organize the real-time applications into independent but cooperating, programs known tasks.
VxWorks Multitasking and Interrupts

Another key facility in real-time systems is hardware interrupt handling.

- Interrupts are the usual mechanism to inform a system of external events.
- It is important to have the fastest possible response to external interrupts.

In VxWorks, *interrupt service routines (ISRs)* run in a special context of their own, outside any task’s context.
VxWorks Task Context

A task’s context includes:

- a thread of execution; that is, the task’s program counter
- the CPU registers and (optionally) floating-point registers
- a stack for dynamic variables and function calls
- I/O assignments for standard input, output, and error
- a delay timer
- a time-slice timer
- kernel control structures
- signal handlers
- debugging and performance monitoring values

In VxWorks, one important resource that is not part of a task’s context is memory address space.
All code executes in a single common address space.
VxWorks Task States

**READY:**
The state of a task that is not waiting for any resource other than the CPU.

**PEND:**
The state of a task that is blocked due to the unavailability of some resource.

**DELAY:** The state of a task that is asleep for some duration.

**SUSPEND:**
The state of a task that is unavailable for execution. This state is used primarily for debugging. Suspension does not inhibit state transition, only task execution. Thus, *pered-suspended* tasks can still unblock and *delayed-suspended* tasks can still be awaken.

**DELAY + S:** The state of a task that is both delayed and suspended.

**PEND + S:** The state of a task that is both pended and suspended.

**PEND + T:** The state of a task that is pended with a timeout value.

**PEND + S + T:** The state of a task that is both pended with a timeout value and suspended.

**state + I:** The state of task specified by *state*, plus an inherited priority.
Task-State Transitions

The highest-priority ready task is executing.

```
ready  --->  pended  semTake() / msgQReceive()
ready  --->  delayed  taskDelay()
ready  --->  suspended  taskSuspend()
pended  --->  ready  semGive() / msgQSend()
pended  --->  suspended  taskSuspend()
delayed  --->  ready  expired delay
delayed  --->  suspended  taskSuspend()
suspended  --->  ready  taskResume() / taskActivate()
suspended  --->  pended  taskResume()
suspended  --->  delayed  taskResume()
```
Wind (VxWorks) Task Scheduling

- The default algorithm in wind is priority-based preemptive scheduling.
- You can also select to use round-robin scheduling for your applications.

Both algorithms rely on the task’s priority. The wind kernel has 256 priority levels, numbered 0 through 255. Priority 0 is the highest and priority 255 is the lowest.

Tasks are assigned a priority when created. You can also change a task’s priority level while it is executing by calling `taskPrioritySet()`.

The ability to change task priorities dynamically allows applications to track precedence changes in the real world.
VxWorks Task Control

VxWorks library taskLib provide routines for task creation and control, as well as for retrieving information about tasks.

Task Creation and Activation

- taskSpawn( ) Spawns (creates and activates) a new task.
- taskInit( )Initializes a new task.
- taskActivate( )Activates an initialized task.

Task Name and ID Routines

- taskName( ) Gets the task name associated with a task ID.
- taskNameToId( ) Looks up the task ID associated with a task.
- taskIdSelf( ) Gets the calling task’s ID.
- taskIdVerify( ) Verifies the existence of a specified task.

Task Information Routines

- taskIdListGet( ) Fills an array with the IDs of all active tasks.
- taskInfoGet( ) Gets information about a task.
- taskPriorityGet( ) Examines the priority of a task.
- taskRegsGet( ) Examines a task’s registers (cannot be used for current task).
VxWorks Task Control

Task-Deletion Routines

- `exit( )` Terminates the calling task and frees memory (task stacks and task control blocks only).
- `taskDelete( )` Terminates a specified task and frees memory (task stacks and task control blocks only).
- `taskSafe( )` Protects the calling task from deletion.
- `taskUnsafe( )` Undoes a `taskSafe( )` (makes the calling task available for deletion).

Task Control Routines

- `taskSuspend( )`Suspends a task.
- `taskResume( )`Resumes a task.
- `taskRestart( )`Restarts a task.
- `taskDelay( )`Delays a task; delay units and resolution in ticks.
- `nanosleep( )`Delays a task; delay units are nanoseconds.
Task Scheduler Control

VxWorks provide routines for task scheduler control.

\text{taskPrioritySet( ) Changes the priority of a task.}

\text{kernelTimeSlice( ) Controls round-robin scheduling.}

Round-robin scheduling is enabled by calling \text{kernelTimeSlice( )}, which takes a parameter for a time slice, or interval. This interval is the amount of time each task is allowed to run before relinquishing the processor to another equal-priority task.

\text{taskLock( ) Disables task rescheduling.}

\text{taskUnlock( ) Enables task rescheduling.}

- The \text{wind} scheduler can be explicitly disabled and enabled on a per-task basis with the routines \text{taskLock( )} and \text{taskUnlock( )}.
- When a task disables the scheduler by calling \text{taskLock( )}, no priority-based preemption can take place while that task is running.
- Note that preemption locks prevent task context switching, but do not lock out interrupt handling.
- Preemption locks can be used to achieve mutual exclusion; however, keep the duration of preemption locking to a minimum.
IPC: Interprocess Communication

OS provides mechanisms so that processes can pass data. Two types of semantics:

- **blocking**: sending process waits for response;
- **non-blocking**: sending process continues.

**IPC styles**

**Shared memory:**
- processes have some memory in common;
- must cooperate to avoid destroying and/or missing any messages.

**Message passing:**
- processes send messages along a communication channel---no common address space.
IPC Styles

Shared memory on a bus:

Message passing
Critical Regions

**Critical region**: section of code that cannot be interrupted by another process.

Examples:
- writing shared memory;
- accessing I/O device.

**Semaphores**

**Semaphore**: OS primitive for controlling access to critical regions.
- Get access to semaphore with $P()$. Perform critical region operations.
- Release semaphore with $V()$. 

Embedded vs. General-Purpose Scheduling

Workstations try to avoid starving processes of CPU access.
  • Fairness = access to CPU.
Embedded systems must meet deadlines.
  • Low-priority processes may not run for a long time.

Priority-driven Scheduling
  • Each process has a priority
  • CPU goes to highest-priority process that is ready
  • Priorities determine the scheduling policy:
    Fixed priority
    Time-varying priorities
Priority-driven Scheduling

Rules:
- each process has a fixed priority (1 highest);
- highest-priority ready process gets CPU;
- process continues until done.

Processes
- P1: priority 1, execution time 10
- P2: priority 2, execution time 30
- P3: priority 3, execution time 20
The Scheduling Problem

• Can we meet all deadlines?
• Must be able to meet deadlines in all cases.
• How much CPU time, we need to meet the deadlines?

Process Initiation

• Periodic process: executes on (almost) every period.
• Aperiodic process: executes on demand.

Analyzing aperiodic process set is harder---must consider worst-case combinations of process activations.
Process Timing Requirements

**Period**: interval between process activations.
**Initiation interval**: reciprocal of period.
**Initiation time**: time at which process becomes ready.
**Deadline**: time at which process must finish.

**Timing violations**
What happens if a process doesn’t finish by its deadline?

- **Hard deadline**: system fails if missed.
- **Soft deadline**: user may notice, but system doesn’t necessarily fail.

**Example**: Space Shuttle software error
A software timing error delayed shuttle’s first launch:
- Primary control system PASS and backup system BFS.
- BFS failed to synchronize with PASS.
- Change to one routine added delay that threw off start time calculation.
- 1 in 67 chance of timing problem.
Process Model

- The application is assumed to consist of a fixed set of processes.
- Processes are completely independent of each other.
- All system's overheads, context-switching times and so on are ignored (i.e. assumed to have zero cost)
- All processes are periodic, with known periods.
- All processes have a deadline equal to their period (that is, each process must complete before it is next released)
- All processes have a fixed worst-case execution time.
CPU Scheduling

CPU scheduling determines which process is going to execute next.

Relevant to Real-time Systems

- CPU scheduler is also known as the dispatcher
- It is invoked on an event that may lead to choose another process for execution:
  - Clock interrupts
  - I/O interrupts
  - Operating system calls and traps
  - Signals

Short-term scheduling
Scheduling Policies

The selection function: It determines which process in the ready queue is selected next for execution.

The decision mode: It specifies the instants in time at which the selection function is exercised

Non-preemptive
- Once a process is in the running state, it will continue until it terminates or blocks itself for I/O.

Preemptive
- Currently running process may be interrupted and moved to the Ready state by the OS.
- Allows for better service since any one process cannot monopolize the processor for very long.
FCFS Scheduling

Service time = Total processor time needed in a (CPU-I/O) cycle

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Service Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

When the current process ceases to execute, the oldest process in the Ready queue is selected.
FCFS: First Come First Served

- Selection function: The process that has been waiting the longest in the ready queue
- Decision mode: Non-preemptive
  Process run until it blocks itself

FCFS Drawbacks
- Process that does not perform any I/O will monopolize the processor.
- Favors CPU-bound processes:
  - I/O-bound processes have to wait until CPU-bound process completes.
  - I/O-bound processes have to wait even when their I/O is completed (poor device utilization).
  - We could have kept the I/O devices busy by giving a bit more priority to I/O bound processes.
Time-Sliced Scheduling

- Known as Round Robin
- Each process runs for a fixed amount of time.
- Processes are run in a round-robin sequence.
- Appropriate for regular multi-programming environments.
- Poor response time performance.
- Need better strategy for real-time system applications.
Round Robin (RR) Scheduling

- Selection function: FCFS
- Decision mode: Preemptive
  - A process is allowed to run until the time slice period has expired
  - Then a clock interrupt occurs and the running process is put on the ready queue.
Round Robin

Time quantum must be substantially larger than the time required to handle the clock interrupt and dispatching. Round Robin favors CPU-bound processes

- I/O bound process uses the CPU for a time less than the time quantum and it is blocked waiting for I/O.
- A CPU-bound process run for full time slice and put back into the ready queue.

Solution: Use Virtual Round Robin

- When an I/O completes, the blocked process is moved to an auxiliary queue that gets preference over the main ready queue.
- A process dispatched from the auxiliary queue runs no longer than the basic time quantum minus the time spent running since it was selected from the ready queue.
**Problem:** Consider the following processes are to be scheduled using FCFS and Round Robin

<table>
<thead>
<tr>
<th>Process</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival Time $T_a$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Service Time $T_s$</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

Perform the analysis for each scheduling algorithm.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>RR, $q = 1$</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCFS</td>
<td>$T_f$</td>
<td>1.00</td>
<td>10.00</td>
<td>11.00</td>
</tr>
<tr>
<td></td>
<td>$T_r$</td>
<td>1.00</td>
<td>9.00</td>
<td>9.00</td>
</tr>
<tr>
<td></td>
<td>$T_r/T_s$</td>
<td>1.00</td>
<td>1.00</td>
<td>9.00</td>
</tr>
<tr>
<td>RR $q = 1$</td>
<td>$T_f$</td>
<td>1.00</td>
<td>18.00</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>$T_r$</td>
<td>1.00</td>
<td>17.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>$T_r/T_s$</td>
<td>1.00</td>
<td>1.89</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Problem. Consider the following processes, A, B, C, D and E that are to be scheduled using, FCFS and Round Robin scheduling techniques with time quantum 1 and 4.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ta</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>Ts</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Where Ta = Process Arrival Time
Ts = Process Service Time

Show a complete schedule for both cases.
Real-time Scheduling Techniques

- Fixed-Priority Scheduling (FPS)
- Earliest Deadline First (EDF)

FPS: Fixed-Priority Scheduling
- This is the most widely used approach.
- Each process has a fixed, *(static)* priority that is computed before execution.
- The runnable processes are executed in the order determined by their priority.
- In real-time systems, the “priority” of a process is derived from its temporal requirements, not its importance to the correct functioning of the system or its integrity.
FPS: Fixed-Priority Scheduling

Rate Monotonic Priority Assignment

• Each process is assigned a (unique) priority based on its period; the shorter the period, the higher the priority
• For two processes i and j:

\[ T_i < T_j \Rightarrow P_i > P_j \]

• An optimal priority assignment means: if any process set can be scheduled (using preemptive priority-based scheduling) with a fixed-priority assignment scheme, then the given process set can also be scheduled with a rate monotonic assignment scheme
• Priority 1 is the lowest (least) priority
## Priority Assignment: An Example

**Period T:** Minimum time between process releases.
**C:** Worst-case computation time (WCET) of the process.
**U:** The utilization of each process (equal to C/T).
**R:** Worst-case response time of the process.
**B:** Worst-case blocking time for the process.
**D:** Deadline of the process.
**N:** Number of process.
**The interference time of the process.**
**Release jitter of the process.**

<table>
<thead>
<tr>
<th>Process</th>
<th>Period, T</th>
<th>Priority, P</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>b</td>
<td>60</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>42</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>105</td>
<td>1</td>
</tr>
<tr>
<td>e</td>
<td>75</td>
<td>2</td>
</tr>
</tbody>
</table>
Utilization-Based Analysis

For D=T process sets, a sufficient but not necessary schedulability test exists.

\[
U \equiv \sum_{i=1}^{N} \frac{C_i}{T_i} \leq N \left(2^{1/N} - 1\right)
\]

<table>
<thead>
<tr>
<th>N</th>
<th>Utilization bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100.0%</td>
</tr>
<tr>
<td>2</td>
<td>82.8%</td>
</tr>
<tr>
<td>3</td>
<td>78.0%</td>
</tr>
<tr>
<td>4</td>
<td>75.7%</td>
</tr>
<tr>
<td>5</td>
<td>74.3%</td>
</tr>
<tr>
<td>10</td>
<td>71.8%</td>
</tr>
</tbody>
</table>
Utilization-Based Analysis

Process Set A

<table>
<thead>
<tr>
<th>Process</th>
<th>Period, T</th>
<th>Computation Time, C</th>
<th>Priority, P</th>
<th>Utilization, U</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>50</td>
<td>12</td>
<td>1</td>
<td>0.24</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
<td>10</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>c</td>
<td>30</td>
<td>10</td>
<td>3</td>
<td>0.33</td>
</tr>
</tbody>
</table>

The combined utilization is 0.82 (or 82%) This is above the threshold for three processes (0.78) and, hence, this process set fails the utilization test.
Time-Line for Process Set A

<table>
<thead>
<tr>
<th>Process</th>
<th>Process Release Time</th>
<th>Process Completion Time</th>
<th>Deadline Met</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- Process a: Preempted from 20 to 40, Executing from 40 to 50
- Process b: Executing from 10 to 40, Preempted from 40 to 50
- Process c: Executing from 0 to 10, Preempted from 10 to 20

Legend:
- Process Release Time
- Process Completion Time
- Deadline Met
- Deadline Missed

- Preempted
- Executing
Utilization-Based Analysis

Process Set B

<table>
<thead>
<tr>
<th>Process</th>
<th>Period $T$</th>
<th>Computation Time $C$</th>
<th>Priority $P$</th>
<th>Utilization $U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>80</td>
<td>32</td>
<td>1</td>
<td>0.400</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
<td>5</td>
<td>2</td>
<td>0.125</td>
</tr>
<tr>
<td>c</td>
<td>16</td>
<td>4</td>
<td>3</td>
<td>0.250</td>
</tr>
</tbody>
</table>

- The combined utilization is 0.775 (or 77.5%)
- This is below the threshold for three processes (0.78) and, hence, this process set will meet all its deadlines.
Utilization-Based Analysis

Process Set C

<table>
<thead>
<tr>
<th>Process</th>
<th>Period T</th>
<th>Computation Time C</th>
<th>Priority P</th>
<th>Utilization U</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>80</td>
<td>40</td>
<td>1</td>
<td>0.50</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
<td>10</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>c</td>
<td>20</td>
<td>5</td>
<td>3</td>
<td>0.25</td>
</tr>
</tbody>
</table>

- The combined utilization is 1.0
- This is above the threshold for three processes (0.78) but the process set will meet all its deadlines.

The Utilization test is said to be sufficient but not necessary
Utilization-based test is neither exact nor general but its O(N)
Earliest Deadline First (EDF) Scheduling

- The runnable processes are executed in the order determined by the absolute deadlines of the processes.
- The next process to run being the one with the shortest (nearest) deadline.
- It is possible to know the relative deadlines of each process e.g. 25ms after release. The absolute deadlines are computed at run time and hence the scheme is described as dynamic.

Value Based (VBS) Scheduling

- If a system can become overloaded then simple static priorities or deadlines are not sufficient; a more adaptive scheme is needed.

This often takes the form of assigning a value to each process and employing an on-line value-based scheduling algorithm to decide which process to run next.
Preemption and Non-Preemption

With priority-based scheduling, a high-priority process may be released during the execution of a lower priority one.

- In a preemptive scheme, there will be an immediate switch to the higher-priority process.
- With non-preemption, the lower-priority process will be allowed to complete before the other executes.
- Preemptive schemes enable higher-priority processes to be more reactive, and hence they are preferred.
- Alternative strategies allow a lower priority process to continue to execute for a bounded time.
- These schemes are known as deferred preemption or cooperative dispatching.
- Schemes such as EDF and VBS can also take on a preemptive or non-preemptive form.
Utilization-based Test for EDF

\[ \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1 \]

A much simpler test

- Superior to FPS; it can support high utilizations.
- However, FPS is easier to implement, as priorities are static.
- EDF is dynamic and requires a more complex run-time system that will have higher overhead.
- It is easier to incorporate processes without deadlines into FPS; giving a process an arbitrary deadline is more artificial.
- It is easier to incorporate other factors into the notion of priority than it is into the notion of deadline.
- During overload situations:
  - FPS is more predictable; Low priority process miss their deadlines first
  - EDF is unpredictable; a domino effect can occur in which a large number of processes miss deadlines
Response-Time Analysis

Task \( i \)'s worst-case response time, \( R \) is calculated first and then checked (trivially) with its deadline.

\[
R_i \leq D_i
\]

\[
R_i = C_i + I_i
\]

where \( I \) is the interference from higher priority tasks

During \( R \), each higher priority task \( j \) will execute a no. of times.

Number of Releases = \[
\left\lceil \frac{R_i}{T_j} \right\rceil
\]

Total interference = \[
\left\lceil \frac{R_i}{T_j} \right\rceil C_j
\]

Ceiling function gives the smallest integer greater than the fractional number on which it acts.

Ceiling of \( 1/3 = 2 \), \( 6/5 = 2 \) And \( 6/3 = 2 \)
Response Time

\[ R_i = C_i + \sum_{j \in hp(i)} \left( \frac{R_i}{T_j} \right) C_j \]

where \( hp(i) \) is the set of tasks with priority higher than task \( i \)

Solve by forming a recurrence relationship:

\[ w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left( \frac{w_i^n}{T_j} \right) C_j \]

The set of values \( w_i^0, w_i^1, w_i^2, ..., w_i^n, ... \) is monotonically non-decreasing

When \( w_i^n = w_i^{n+1} \) the solution to the equation has been found, \( w_i^0 \) must not be greater than \( R_i \) (e.g. 0 or \( C_i \))
Response Time Calculation Algorithm

\[ \text{for } i \text{ in } 1..N \text{ loop -- for each process in turn} \]
\[ n := 0 \]
\[ w_i^n := C_i \]
\[ \text{loop} \]
\[ \text{calculate new } w_i^{n+1} \]
\[ \text{if } w_i^{n+1} = w_i^n \text{ then} \]
\[ R_i = w_i^n \]
\[ \text{exit value found} \]
\[ \text{end if} \]
\[ \text{if } w_i^{n+1} > T_i \text{ then} \]
\[ \text{exit value not found} \]
\[ \text{end if} \]
\[ n := n + 1 \]
\[ \text{end loop} \]
\[ \text{end loop} \]
# Response Time Calculation Example

## Process Set D

<table>
<thead>
<tr>
<th>Process</th>
<th>Period, T</th>
<th>Computation Time, C</th>
<th>Priority, P</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>7</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>12</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>20</td>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
R_a = 3
\]

\[
w_b^0 = 3
\]

\[
w_b^1 = 3 + \left\lfloor \frac{3}{7} \right\rfloor 3 = 6
\]

\[
w_b^2 = 3 + \left\lfloor \frac{6}{7} \right\rfloor 3 = 6
\]

\[
R_b = 6
\]
Response Time Calculation

Process $c$

\[ w_c^0 = 5 \]
\[ w_c^1 = 5 + \left[ \frac{5}{7} \right] 3 + \left[ \frac{5}{12} \right] 3 = 11 \]
\[ w_c^2 = 5 + \left[ \frac{11}{7} \right] 3 + \left[ \frac{11}{12} \right] 3 = 14 \]
\[ w_c^3 = 5 + \left[ \frac{14}{7} \right] 3 + \left[ \frac{14}{12} \right] 3 = 17 \]
\[ w_c^4 = 5 + \left[ \frac{17}{7} \right] 3 + \left[ \frac{17}{12} \right] 3 = 20 \]
\[ w_c^5 = 5 + \left[ \frac{20}{7} \right] 3 + \left[ \frac{20}{12} \right] 3 = 20 \]
\[ R_c = 20 \]
Process Set C

<table>
<thead>
<tr>
<th>Process</th>
<th>Period, T</th>
<th>Computation Time, C</th>
<th>Priority, P</th>
<th>Response Time, R</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>80</td>
<td>40</td>
<td>1</td>
<td>80</td>
</tr>
<tr>
<td>b</td>
<td>40</td>
<td>10</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>c</td>
<td>20</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

- The combined utilization is 1.0.
- This was above the utilization threshold for three processes (0.78) therefore it failed the test.
- The response time analysis shows that the process set will meet all its deadlines.
- RTA is necessary and sufficient.

If the process set passes the test they will meet all their deadlines; if they fail the test then, at run-time, a process will miss its deadline. (unless computation time estimations themselves turn out to be pessimistic)
Worst-Case Execution Time – WCET

- Obtained by either measurement or analysis
- The problem with measurement is that it is difficult to be sure when the worst case has been observed.
- The drawback of analysis is that an effective model of the processor (including caches, pipelines, memory wait states and so on) must be available.

Most analysis techniques involve two distinct activities.
- The first takes the process and decomposes its code into a directed graph of basic blocks.
- These basic blocks represent straight-line code.
- The second component of the analysis takes the machine code corresponding to a basic block and uses the processor model to estimate its worst-case execution time.
- Once the times for all the basic blocks are known, the directed graph can be collapsed.
WCET Analysis

Need Semantic Information

```java
for I in 1.. 10 loop
    if Cond then
        -- basic block of cost 100
    else
        -- basic block of cost 10
    end if;
end loop;
```

- Simple cost $10 \times 100$ (+overhead), say 1005.
- But if `Cond` only true on 3 occasions then cost is 375
Real-time Scheduling Exercises

**Exercise-1:**
Consider three processes P, Q and S. P has a period of 100msec in which it requires 30msecs of processing. The corresponding values for Q and S are (6, 1) and (25, 5) respectively. Assume that P is the most important process in the system, followed by Q and then S.

1. What is the behavior of the scheduler if priority is based on importance?
2. What is the process utilization of P, Q and S.
3. How should the process be scheduled so that all deadlines are met.
4. Illustrate one of the schemes that allows these processes to be scheduled.

**Exercise-2:**
Add a fourth process R, to the set of processes given in Exercise-1. Failure of this process will not lead to safety being undermined. R has a period of 50ms, but has a processing requirement that is data dependent and varies from 5 to 25 ms. Discuss how this process should be integrated with P, Q and S.
Hard and Soft Real-time Processes

**Hard Real-time Process:** The deadline must not be missed.

**Soft Real-time Process:** The application is tolerant of missed deadlines.

- In many situations the WCET (worst-case execution time) figures for sporadic processes are considerably higher than the averages.

- Measuring schedulability with worst-case figures may lead to very low processor utilizations.

- Interrupts often arrive in bursts e.g. an abnormal sensor reading may lead to significant additional computation.
Sporadic Processes

• A Sporadic process is that which has hard real-time applications.

• Sporadic processes have a minimum inter-arrival time.

• They also require $D < T$

• The response time algorithm for fixed priority-scheduling works perfectly for values of $D$ less than $T$ as long as the stopping criteria becomes $W_{i}^{n+1} > D_{i}$

• It also works perfectly well with any priority ordering, $hp(i)$ always gives the set of higher-priority processes
Hard/Soft Process Scheduling Guidelines

**Rule 1** — all processes should be schedulable using average execution times and average arrival rates.

**Rule 2** — all hard real-time processes should be schedulable using worst-case execution times and worst-case arrival rates of all processes (including soft)

- A consequent of Rule 1 is that there may be situations in which it is not possible to meet all current deadlines
- This condition is known as a *transient overload*
- Rule 2 ensures that no hard process will miss its deadline
- If Rule 2 gives rise to unacceptably low utilizations for “normal execution” then action must be taken to reduce the worst-case execution times (or arrival rates)
Aperiodic Processes

- Aperiodic processes have soft real-time jobs.
- They do not have minimum inter-arrival times.
- Can run aperiodic processes at a priority below the priorities assigned to hard processes, therefore, they cannot steal, in a pre-emptive system, resources from the hard processes.
- This does not provide adequate support to soft processes, which will often miss their deadlines.
- To improve the situation for soft processes, a server (sporadic) can be employed.
- Servers protect the processing resources needed by hard processes but otherwise allow soft processes to run as soon as possible.
- POSIX support Sporadic Servers
Process Sets with D < T

- For D = T, Rate Monotonic priority ordering is optimal.
- For D < T, (DMPO) Deadline Monotonic Priority Ordering is optimal.

\[ D_i < D_j \Rightarrow P_i > P_j \]

D < T Example Process Set

<table>
<thead>
<tr>
<th>Process</th>
<th>Period T</th>
<th>Deadline D</th>
<th>Computation Time, C</th>
<th>Priority P</th>
<th>Response Time, R</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>20</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>15</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>10</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>20</td>
<td>20</td>
<td>3</td>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

Proof of “DMPO is Optimal” is given in the text
Deadline Scheduling Exercises

**Exercise-1:**
Consider two jobs, A and B, in a deadline scheduling system. The deadline for A is before the deadline for B. Explain why we should run A before B, that is, show that if running A then B fails to meet some deadline then running B before A will also fail to meet some deadline.

**Exercise –2:**
Consider a set of 5 aperiodic tasks whose execution profiles are given below. Develop the scheduling diagram of these processes employing EDF and FCFS.

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Execution Time</th>
<th>Starting Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>C</td>
<td>40</td>
<td>20</td>
<td>60</td>
</tr>
<tr>
<td>D</td>
<td>50</td>
<td>20</td>
<td>80</td>
</tr>
<tr>
<td>E</td>
<td>60</td>
<td>20</td>
<td>70</td>
</tr>
</tbody>
</table>
Process Interactions and Blocking

- If a process is suspended waiting for a lower-priority process to complete some required computation then the priority model is, in some sense, being undermined.
- The process is said to suffer *priority inversion*.
- If a process is waiting for a lower-priority process, the process is said to be *blocked*.
- Dynamic priorities can vary during execution.

One has to avoid Priority Inversion.

**Bounded Priority Inversion**
- Duration is not longer than that of the critical section where the lower-priority process owns the resource.

**Unbounded Priority Inversion**
- Occurs when a *third* (medium-priority) process preempts the low-priority process during the inversion for an indefinite time.
Priority Inversion

An extreme example of priority inversion, consider the executions of four periodic processes: a, b, c and d; and two resources: Q and V

Example of Priority Inversion

<table>
<thead>
<tr>
<th>Process</th>
<th>Priority</th>
<th>Execution Sequence</th>
<th>Release Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>EQQQQE</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>EE</td>
<td>2</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>EVVE</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>EEQVE</td>
<td>4</td>
</tr>
</tbody>
</table>
Example of Priority Inversion

Process

- d
- c
- b
- a

Process Completion Time
Deadline Met

Process Release Time

- Executing
- Preempted
- Executing with Q locked
- Blocked
- Executing with V locked
Priority Inheritance

If process \textbf{a} is blocking the process \textbf{d}, then it runs with the priority of \textbf{d}.

Process

\begin{itemize}
  \item \textbf{d}
  \item \textbf{c}
  \item \textbf{b}
  \item \textbf{a}
\end{itemize}

Process Release Time

Process Completion Time

Deadline Met

0  2  4  6  8  10  12  14  16  18
Calculating Blocking

- If a process has $m$ critical sections that can lead to its blocking then the maximum number of times it can be blocked is $m$.
- If $B$ is the maximum blocking time and $K$ is the number of critical sections, the process $i$ has an upper bound on its blocking given by: $B_i = \sum_{k=1}^{K} \text{usage}(k,i)C(k)$

Response Time and Blocking:

$$R_i = C_i + B_i + I_i$$

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left[ \frac{R_i}{T_j} \right] C_j$$

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left[ \frac{w_i^n}{T_j} \right] C_j$$
Priority Ceiling Protocols

**OCPP:** Original ceiling priority protocol

**ICPP:** Immediate ceiling priority protocol

**OCPP**

- Each process has a static default priority assigned (perhaps by the deadline monotonic scheme)
- Each resource has a static ceiling value defined, this is the maximum priority of the processes that use it.
- A process has a dynamic priority that is the maximum of its own static priority and any it inherits due to it blocking the higher-priority processes.
- A process can only lock a resource if its dynamic priority is higher than the ceiling of any currently locked resource. (excluding any that it has already locked itself)
OCPP Inheritance

- Executing
- Executing with Q locked
- Executing with V locked

Process release and completion times are shown with timelines for processes a, b, c, and d. The timeline also indicates whether the deadline was met for each process.
ICPP

- Each process has a static default priority assigned (perhaps by the deadline monotonic scheme).
- Each resource has a static ceiling value defined, this is the maximum priority of the processes that use it.
- A process has a dynamic priority that is the maximum of its own static priority and the ceiling values of any resources it has locked.
- As a consequence, a process will only suffer a block at the very beginning of its execution.
- Once the process starts actually executing, all the resources it needs must be free; if they were not, then some process would have an equal or higher priority and the process's execution would be postponed.
ICPP Inheritance

- Executing
- Executing with Q locked
- Executing with V locked

Process Completion Time
Deadline Met
Process Release Time

Process
- d
- c
- b
- a

Timeline:
0 2 4 6 8 10 12 14 16 18

© G. Khan
Embedded Computer Systems–EE8205: Multitasking & Real-time Scheduling
Page: 92
OCPP versus ICPP

The worst-case behavior of the two ceiling schemes is identical (from a scheduling view point)

- A high-priority process can be blocked at most once during its execution by lower-priority processes
- Deadlocks are prevented.
- Transitive blocking is prevented.
- Ensure mutual exclusive access to resources (by protocol itself)

There are some points of difference:

- ICPP is easier to implement than the original (OCPP) as blocking relationships need not be monitored
- ICPP leads to less context switches as blocking is prior to first execution
- ICPP requires more priority movements as this happens with all resource usage
- OCPP changes priority only if an actual block has occurred.

ICPP is called Priority Protect Protocol in POSIX
Mars Pathfinder Suffered Unbounded Priority Inversion

Low-priority Meteorological Process:
  Acquired the (shared) bus.

Medium-priority, Long-running, Communications Process:
  Woke up and preempted the meteorological thread.

High-priority Bus Management Process:
  Woke up and was blocked because it couldn't acquire the bus;
  When it couldn't meet its deadline it reinitialized the computer via a hardware reset.
Mars Pathfinder
Duration of an Unbounded Priority Inversion

Limiting the duration of unbounded priority inversion prevents low-priority process from being preempted by the medium-priority processes during the priority inversion.

Technique: Manipulate process priorities at run-time.

Scheduling: Processes with higher priority are scheduled to run first.

Objective: Assign priorities in such a way that all outputs are computed before their deadlines.

- Deadline-Driven Assignment: Assign highest priorities to processes with shortest deadlines.
- Rate Monotonic Assignment: Assign highest priorities to processes that run most frequently without regard to deadlines.
Modified Process Model

Until Now:

- Deadlines can be less than period (D < T)
- Sporadic and aperiodic processes, as well as periodic processes, can be supported
- Process interactions are possible, with the resulting blocking being factored into the response time equations.

Extensions to the Original Model

- Cooperative Scheduling
- Release Jitter
- Arbitrary Deadlines
- Fault Tolerance
- Offsets
- Optimal Priority Assignment
Cooperative Scheduling

- True preemptive behavior is not always acceptable for safety-critical systems
- Cooperative or deferred preemption splits processes into slots
- Mutual exclusion is via non-preemption
- The use of deferred preemption has two important advantages
  - It increases the schedulability of the system, and it can lead to lower values of $C$(computation time). With deferred preemption, no interference can occur during the last slot of execution.

Let the execution time of the final block be

$$w_i^{n+1} = B_{MAX} + C_i - F_i + \sum_{j \in hp(i)} \left[ \frac{w_i^n}{T_j} \right] C_j$$

After the solution converge i.e. $w_i^n = w_i^{n+1}$

The response time is given by: $R_i = w_i^n + F_i$
Arbitrary Deadlines

To cater for the situations where \( D \) (and hence potentially \( R \)) > \( T \)

\[
w_i^{n+1}(q) = B_i + (q + 1)C_i + \sum_{j \in hp(i)} \left( \frac{w_i^n(q)}{T_j} \right) C_j
\]

\[
R_i(q) = w_i^n(q) - qT_i
\]

- The number of releases is bounded by the lowest value of \( q \) for which the following relation is true:
  \( R_i(q) \leq T_i \)
- The worst-case response time is then the maximum value found for each \( q \):
  \[
  R_i = \max_{q=0,1,2,...} R_i(q)
  \]
Fault Tolerance

- Fault tolerance via either forward or backward error recovery always results in extra computation.
- This could be an exception handler or a recovery block.
- In a real-time fault tolerant system, deadlines should still be met even when a certain level of faults occur.
- This level of fault tolerance is known as the fault model.
- If the extra computation time that results from an error in process \( i \) is \( C^f_i \),

\[
R_i = C_i + B_i + \sum_{j \in \text{hp}(i)} \left( \frac{R_i}{T_j} \right) C_j + \max_{k \in \text{hp}(i)} C^f_k
\]

where \( \text{hp}(i) \) is set of processes with priority equal to or higher than \( i \).
Fault Tolerance

If $F$ is the number of faults allows

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left( \frac{R_i}{T_j} \right) C_j + \max_{k \in hep(i)} FC^f_k$$

If there is a minimum arrival interval

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left( \frac{R_i}{T_j} \right) C_j + \max_{k \in hep(i)} \left( \left[ \frac{R_i}{T_f} \right] C_k^f \right)$$
Offsets

So far assumed all processes share a common release time (critical instant)

<table>
<thead>
<tr>
<th>Process</th>
<th>T</th>
<th>D</th>
<th>C</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>20</td>
<td>10</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>c</td>
<td>20</td>
<td>12</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>

With offsets

<table>
<thead>
<tr>
<th>Process</th>
<th>T</th>
<th>D</th>
<th>C</th>
<th>O</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>b</td>
<td>20</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>c</td>
<td>20</td>
<td>12</td>
<td>4</td>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>
Non-Optimal Analysis

- In most realistic systems, process periods are not arbitrary but are likely to be related to one another.
- When two processes have a common period. In these situations it is easy to give one process an offset (of $T/2$) and to analyze the resulting system using a transformation technique that removes the offset - and, hence, critical instant analysis applies.

Last page example: processes $b$ and $c$ (having the offset of 10) are replaced by a single notional process with period 10, computation time 4, deadline 10 but no offset
Non-Optimal Analysis

The notional process, \( n \) has two important properties.

- If it is schedulable (when sharing a critical instant with all other processes) then the two real process will meet their deadlines when one is given the half period offset
- If all lower priority processes are schedulable when suffering interference from the notional process (and other high-priority processes) then they will remain schedulable when the notional process is replaced by two real processes (one with the offset).

These properties follow from the observation that the notional process always uses more (or equal) CPU time than the two real process.

<table>
<thead>
<tr>
<th>Process</th>
<th>T</th>
<th>D</th>
<th>C</th>
<th>O</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>
Notional Process Parameters

\[ T_n = \frac{T_a}{2} = \frac{T_b}{2} \]

\[ C_n = \text{Max}(C_a, C_b) \]

\[ D_n = \text{Min}(D_a, D_b) \]

\[ P_n = \text{Max}(P_a, P_b) \]

The above parameters can be extended to more than two processes
Priority Assignment

*Theorem:* If process p is assigned the lowest priority and is feasible then, if a feasible priority ordering exists for the complete process set, an ordering exists with process p assigned the lowest priority.

```plaintext
procedure Assign_Pri (Set: in out Process_Set;
                     N: Natural; Ok: out Boolean) is
begin
  for K in 1..N loop
    for Next in K..N loop
      Swap(Set, K, Next);
      Process_Test(Set, K, Ok);
      exit when Ok;
    end loop;
    exit when not Ok; -- failed to find a schedulable process
  end loop;
end Assign_Pri;
```
Dynamic Systems and Online Analysis

- There are dynamic soft real-time applications in which arrival patterns and computation times are not known a priori.

- Although some level of off-line analysis may still be applicable, this can no longer be complete and hence some form of on-line analysis is required.

- The main task of an on-line scheduling scheme is to manage any overload that is likely to occur due to the dynamics of the system's environment.

- EDF is a dynamic scheduling scheme that is an optimal.

- During transient overloads EDF performs very badly. It is possible to get a cascade effect in which each process misses its deadline but uses sufficient resources to result in the next process also missing its deadline.
Admission Schemes

To counter this detrimental domino effect many on-line schemes have two mechanisms:

- An admissions control module that limits the number of processes that are allowed to compete for the processors, and
- An EDF dispatching routine for those processes that are admitted.

• An ideal admissions algorithm prevents the processors getting overloaded so that the EDF routine works effectively.
Values

• If some processes are to be admitted, whilst others rejected, relative importance of each process must be known.
• This is usually achieved by assigning value.
• Values can be classified as:
  Static: a process always has the same value whenever it is released.
  Dynamic: the process's value can only be computed at the time the process is released (because it is dependent on either environmental factors or the current state of the system)
  Adaptive: here the dynamic nature of the system is such that the value of the process will change during its execution.

• To assign static values requires the domain specialists to articulate their understanding of the desirable behavior of the system.
POSIX

- POSIX supports priority-based scheduling, and has options to support priority inheritance and ceiling protocols
- Priorities may be set dynamically
- Within the priority-based facilities, there are four policies:
  - FIFO: a process/thread runs until it completes or it is blocked
  - Round-Robin: a process/thread runs until it completes or it is blocked or its time quantum has expired
  - Sporadic Server: a process/thread runs as a sporadic server
  - OTHER: an implementation-defined
- For each policy, there is a minimum range of priorities that must be supported; 32 for FIFO and round-robin
- The scheduling policy can be set on a per process and a per thread basis
POSIX

• Threads may be created with a *system contention* option, in which case they compete with other system threads according to their policy and priority.

• Alternatively, threads can be created with a *process contention* option where they must compete with other threads (created with a process contention) in the parent process.
  - It is unspecified how such threads are scheduled relative to threads in other processes or to threads with global contention.

• A specific implementation must decide which to support.

• POSIX also allows:
  - Priority inheritance to be associated with mutexes (priority protected protocol= ICPP)
  - Message queues to be priority ordered
  - Functions for dynamically getting and setting a thread's priority
  - Threads to indicate whether their attributes should be inherited by any child thread they create.
Sporadic Server in POSIX

- A sporadic server assigns a limited amount of CPU capacity to handle events, has a replenishment period, a budget, and two priorities.
- The server runs at a high priority when it has some budget left and a low one when its budget is exhausted.
- When a server runs at the high priority, the amount of execution time it consumes is subtracted from its budget.
- The amount of budget consumed is replenished at the time the server was activated plus the replenishment period.
- When its budget reaches zero, the server's priority is set to the low value.
Critical Task Scheduling Exercise

Real-time system designers wish to run a mixture of safety-critical, mission-critical and non-critical periodic and sporadic tasks on the same processor. They are using preemptive priority-based scheduling and have used the response-time analysis equation to predict that all tasks meet their deadlines.

Give reasons why the system might nevertheless fail to meet its deadlines at run-time.
What enhancement could be provided to the runtime support system to help eliminate the problem?