Unit III - Process and Operating Systems

Multiple Tasks and multiprocesses.
- A task is a functional description of a connected set of operations. It is a collection of processes.
- A process is a unique execution of a program. Several copies of a program may run simultaneously or at different times. A process has its own state—registers and memory. The operating system manages processes.
- Multiple tasks means multiple processes.
- Reactive systems respond to external events and require real-time response. It may require a chain reaction among multiple processors.
- Multirate systems—Tasks may be synchronous or asynchronous. Synchronous tasks may recur at different rates.
- Real Time Systems—perform a computation based on external timing constraints.
Deadline Frequency - Periodic, Aperiodic

Deadline Types:
1. Hard → failure to meet deadline causes system failure
2. Soft → failure to meet deadline causes degraded response
3. Firm → late response is useless but some late response can be tolerated.

Timing Specifications on processes:
1. Release time → Time at which process becomes ready.
2. Deadline → Time at which process must finish.

Period → Interval between process activations.
Rate → Reciprocal of period.

Task graphs:
- Task graph shows data/control dependencies between processes.
  (Data dependency - execute in certain order).
- Task-connected set of processes.
- Task set - One or more Tasks.
Task graph assumes that all processes in each task run at the same rate, tasks at the do not communicate. In reality, some amount of inter-task communication is necessary. It’s hard to require immediate response for multi-rate communication.

**Process Execution Characteristics**

- Process execution time $T_i$ (in absence of preemption) - possible time units - seconds, clock cycles.
- Sources of variation - Data dependencies, memory system, CPU pipeline.
- CPU utilization - fraction of the CPU that is doing useful work. Often calculated assuming no scheduling overhead.

\[
\text{Utilization} = \frac{\text{CPU time for useful work}}{\text{Total available CPU time}}
\]

\[
\frac{T_i}{T}
\]
State of a process

A process can be in one of 3 states,
1. Execution on the CPU
2. Ready to run
3. Waiting for data

Hyperperiod - least common multiple of the task periods.
- Must look at the hyperperiod schedule to find all task interactions.
- Hyperperiod can be very long if task periods are not chosen carefully.

Cyclostatic / TDMA
- Schedule in time slots.
- Time slots may be equal size or unequal.

TDMA assumption - Schedule based on LCM of the process periods.
- Always have same CPU utilization.
- But can't handle unexpected loads.
Round Robin
- Schedule process only if ready.
- Start round-robin again after finishing a round.
- Schedule based on LCM of the process periods.
- Best done with equal time slots for processes.
- Simple scheduler.
- Low scheduling overhead.
- It can be implemented in hardware.
- Can bound maximum CPU load.
- May leave unused CPU cycles.
- It can be adapted to handle unexpected load.
- Use time slots at end of period.

Schedulability and overhead
- Scheduling process consumes CPU time.
- Not all time is available for processes.
- Scheduling overhead must be taken into account for exact schedule.
Running periodic process

- Need to code to control execution of processes
- Simplest implementation has one loop:
  \[ \text{while (Time)} \]
  \[ \quad \text{p1();} \]
  \[ \quad \text{p2();} \]
  \[ \quad \]Timed loop implementation.
- Encapsulate set of all processes in a single function that implements the task set.
- Use time to control execution of the task.
- No control over timing of individual processes.

```c
void p1() {
    p1();
    p2();
}
```
Multiple times implementation.
- Each task has its own function & timer.
- May not have enough times to implement all the rates.

```c
void pA(int); /* rate A */
p1();
p2();
p3();
p4();
p5();
```

```c
void B(int); /* rate B */
p2();
p4();
p5();
```

Time & Counter Implementation.
- Use a software count to divide the timer.
- Only works for clean multiples of the timer period.
- All of these implementations are inadequate.
- Need better control over timing.
Operating Systems

The operating system controls resources:
- who gets the CPU.
- when I/O takes place
- how much memory is allocated.

The most important resource is the CPU itself. CPU access controlled by the scheduler.

Process State can be in one of 3 states,
1. Execution
2. Ready
3. Waiting.

Operating system structure

OS needs to keep track of:
1. process priorities
2. scheduling state
3. process activation record.

Processes may be created:
1. statically before system starts,
2. dynamically during execution.

Priority-driven Scheduling.

- Each process has a priority.
- CPU goes to highest priority process that is ready.

Priorities determine scheduling policy.
1. fixed & time varying priorities.
Priority driven scheduling rules:
1. Each process has a fixed priority.
2. Highest-priority ready process gets CPU.

Process initiation disciplines:
Periodic process: executes on almost every period.
Aperiodic process: executes on demand.

Timing requirements on processes:
→ 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.

Interprocess Communication:
- OS provides mechanisms so that processes can pass data.
- There are 2 types of semantics:
  1. Blocking - sending process waits for response.
  2. Non-blocking - sending process continues.
IPC Styles

1. Shared Memory
   - Processes have some memory in common.
   - Must cooperate to avoid destroying/missing messages.

2. Message passing
   - Processes send messages along a communication channel - no common address space.
   - Shared memory on a bus

Race Condition in shared memory

- Problem when two CPUs try to write in the same location.
  - CPU1 reads flag and sees 0.
  - CPU2 reads flag 0 and sees 0.
  - CPU1 sets flag to one and writes location.
  - CPU2 sets flag to one and overwrites location.
This problem can be solved with an atomic test-and-set-single bus operation reads memory location, tests it, writes it.

Critical regions - section of code that cannot be interrupted by another process, e.g., writing shared memory, accessing I/O device.

Semaphores - OS primitive for controlling access to critical regions.

Protocol: 1) Get access to semaphore with P(); 2) Perform critical region operation; 3) Release semaphore with V().

Message passing on a network.

Other OS functions:
- Date/time
- File system
- Networking
- Security.
Evaluating RTOS performance

Assumptions
1. Context switch costs no CPU time
2. We know the exact execution time of processes.

Process execution time is not constant.

Processes and caches
- Processes can cause additional caching problems
- Even if individual processes are well-behaved, processes may interfere with each other.
- Worst-case execution time with bad behavior is usually much worse than execution time with good cache behavior.

Power Optimization Strategies for Processes

Power Management - determining how system resources are scheduled/used to control power consumption.
OS can manage for power just as it manages for time.
As reduces power by shutting down units or may have partial shutdown modes. 

**Power management and performance.**
- Entering power-down mode consumes energy & time.
- Leaving power-down mode consumes energy & time.

**Simple power management policies**

- **Request-driven:** power up once request is received. Adds delay to response.
- **Predictive shutdown:** try to predict how long you have before next request. 
  - May start up in advance of request in anticipation of a new request.
  - If you predict wrong, you will incur additional delay while starting up.

**Probabilistic shutdown**
- Assume service requests are probabilistic.
- Optimize expected values for power consumption & response time.
- For simple probabilistic models - shut down after time $T_{on}$, turn back on after waiting for $T_{off}$. 
Advanced Configuration & Power Interface (ACPI) - Open standard for power management services.

- applications
- device drivers
- OS kernel
- ACPI BIOS
- Hardware platform

ACPI global power states:

- G0: working state
- G1: sleeping state
- G2: soft off
- Lx31 - low wake-up latency with no loss of context
- Lx32 - low latency with loss of CPU/cache state
- Lx33 - low latency with loss of all state except memory
- Lx34 - lowest power state with all devices off
- G3: Mechanical off.
Context Switching

- It is the procedure of storing the state of an active process for the CPU when it has to start executing a new one. For example, process A with its address space and stack is currently being executed by the CPU and there is a system call to jump to a higher priority process B, then the CPU needs to remember the current state of the process A, so that it can suspend its operation, begin executing the new process B & when done, return to its previously executing process A.

- They can be software or hardware governed depending upon the CPU architecture.

Multiprocessor

- A multiprocessor is a tightly coupled computer system having 2 or more
processing units (Multiple processors) each sharing main memory & peripherals, in order to simultaneously process programs.
Building Basic RTOS

- Preemptive OS
- Priorities
Preemption

• An alternative to C function call to control execution.
• Kernel – Part of OS, that determines which process is running.
• Time Quantum – length of the timer period.
• Context Switching
• Process control block – Data Structure that holds the state of the process.
## Priorities

<table>
<thead>
<tr>
<th>Process</th>
<th>Priority</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>20</td>
</tr>
</tbody>
</table>

![Diagram showing priority scheduling with processes P2, P1, and P3 with release and execution times.](image-url)
Processes & Context

[Diagram showing processes and context with labels for timer, vPreemptiveTick, portSAVE_CONTEXT, portRESTORE_CONTEXT, vTaskSwitchContext, task 1, and task 2.]
Processes & Object oriented Design

```
processClass 1
myAttributes
myOperations()
Signals
start
resume

p1: processClass1
  a: rawMsg
  w: wrapperClass
    ahat: fullMsg
    master: masterClass
```
Example Real Time Operating Systems
Unix

- Unix developed in 1960’s at Bell Laboratories to support text processing.
- POSIX is a standard version of Unix.
- Linux is an open-source POSIX-compliant operating system.
  - Linux versions have been developed to improve real-time responsiveness.
POSIX process creation

- `fork()` makes two copies of executing process.
- Child process identifies itself and overlays new code.

```c
if (childid == 0) {
    /* must be child */
    execv("mychild", childargs);
    perror("execv");
    exit(1);
}
else { /* is the parent */
    parent_stuff();
    wait(&cstatus);
    exit(0);
}
```
POSIX real-time scheduling

- Processes may run under different scheduling policies.
- `_POSIX_PRIORITY_SCHEDULING` resource supports real-time scheduling.
- SCHED_FIFO supports RMS.

```c
int i, my_process_id;
struct sched_param
    my_sched_params;
...

i =
    sched_setscheduler(my_process_id,SCHED_FIFO,&sched_params);
```
POSIX interprocess communication

• Supports counting semaphores in _POSIX_SEMAPHORES.
• Supports shared memory.

i = sem_wait(my_semaphore); /* P */
/* do useful work */
i = sem_post(my_semaphore); /* V */
/* sem_trywait tests without blocking */
i = sem_trywait(my_semaphore);
POSIX pipes

- Pipes directly connect programs.
- `pipe()` function creates a pipe to talk to a child before the child is created.

```c
if (pipe(pipe_ends) < 0) {
    perror("pipe");
    break;
}
childid = fork();
if (childid == 0) {
    childargs[0] = pipe_ends[1];
    execv("mychild", childargs);
    perror("execv");
    exit(1);
} else { ... }
```
POSIX message queues

• Supports message queues under _POSIX_MESSAGE_PASSING

• `mq_open()` creates named queue.

```c
myq = mq_open("/q1", O_CREAT | O_RDWR, S_IRWXU, &mq_attr);
...
if (mq_send(myq, data, len, priority) < 0) { /* error */ }

nbytes =
    mq_receive(myq, rcvbuf, MAXLEN, &prio);
```
Windows CE

• Used in smart phones, electronic instruments, etc.
• Runs on multiple instruction set architectures.
• Some aspects, such as interrupt structure, depend on hardware.
Windows CE layer structure

- applications
- embedded shell
- WinCE shell services
- remote connectivity
- Win32 APIs
  - kernel library
  - GWES
  - device manager
  - file manager
  - TCP/IP
- OEM hardware
  - OAL bootloader
  - drivers
  - device drivers
  - file drivers
  - network drivers
Windows CE OAL architecture

- OEM adaptation layer interfaces to hardware.
- Provides real-time clock, power management, interrupts, debugging, etc.
- Board support package includes OAL and drivers.
Windows CE memory space

- Flat 32-bit virtual address space.
- Addresses can be statically or dynamically mapped.
- Flash or disk can be used for backing store.
Windows CE user address space

- Bottom 1 GB holds user code, data, stack, and heap.
- Top 1 GB used for system heap, DLLs, etc.
Windows CE threads and drivers

- Two units of execution: thread and driver.
- Threads are defined by files.
- Drivers are defined by DLLs.
- Process can have multiple threads, scheduled directly by OS.
Windows CE scheduling

• Threads have integer priorities.
  – 0 is highest priority, 255 is lowest.
  – 248-255 are for non-real-time threads.

• Two policies: thread may run until end of its quantum; thread runs until higher-priority thread is ready.

• Supports priority inheritance.
Windows CE interrupt handling

• Interrupt service handler (ISH) provides initial response to interrupt.

• Interrupt service routine (ISR) handles the interrupt in the kernel with interrupts off.

• Interrupt service thread (IST) runs on OAL and performs most of the work.
Windows CE interrupt sequence diagram