What's An OS?

- Provides environment for executing programs
- Process abstraction for multitasking/concurrency
  - scheduling
- Hardware abstraction layer (device drivers)
- File systems
- Communication

- Do we need an OS?
  - Not always
- Simplest approach: cyclic executive

```
loop
  do part of task 1
  do part of task 2
  do part of task 3
end loop
```

Cyclic Executive

- Advantages
  - Simple implementation
  - Low overhead
  - Very predictable

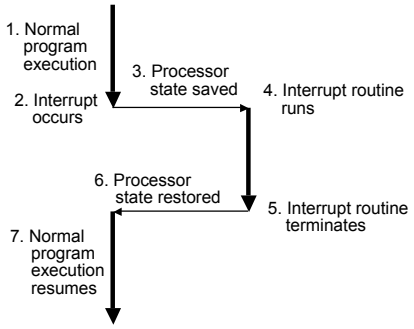
- Disadvantages
  - Can't handle sporadic events
  - Everything must operate in lockstep
  - Code must be scheduled manually

Interrupts

- Some events can't wait for next loop iteration
  - communication channels
  - transient events
- A solution: Cyclic executive plus interrupt routines
- Interrupt: environmental event that demands attention
  - example: "byte arrived" interrupt on serial channel
- Interrupt routine: piece of code executed in response to an interrupt
- Most interrupt routines:
  - copy peripheral data into a buffer
  - indicate to other code that data has arrived
  - acknowledge the interrupt (tell hardware)

  - longer reaction to interrupt performed outside interrupt routine
  - e.g., causes a process to start or resume running

Handling An Interrupt



Cyclic Executive Plus Interrupts

- Works fine for many signal processing applications
- 56001 has direct hardware support for this style
- Insanely cheap, predictable interrupt handler:
  - when interrupt occurs, execute a single user-specified instruction
  - this typically copies peripheral data into a circular buffer
  - no context switch, no environment save, no delay
- Drawbacks:
  - main loop still running in lockstep
  - programmer responsible for scheduling
  - scheduling static
  - sporadic events handled slowly

Cooperative Multitasking

- A cheap alternative
- Non-preemptive
- Processes responsible for relinquishing control
- Examples: original Windows, Macintosh
- A process had to periodically call get_next_event() to let other processes proceed
- OS responsible for deciding which task to run next
- Drawbacks:
  - programmer had to ensure this was called frequently
  - an errant program would lock up the whole system
- Alternative: preemptive multitasking

Cooperative Multitasking

```
void main() {
  Event e;
  while ((e=get_next_event()) != QUIT) {
    switch (e) {
      case CLICK: /* ... */ break;
      case DRAG: /* ... */ break;
      case DOUBLECLICK: /* ... */ break;
      case KEYDOWN: /* ... */ break;
    /* ... */
    }
  }
}
```

Concurrency Provided By OS

- Basic philosophy:

  Let the operating system handle scheduling, and let the programmer handle function

- Scheduling and function usually orthogonal
- Changing the algorithm would require a change in scheduling

Batch Operating Systems

- Original computers ran in batch mode:
  - submit job & its input
  - job runs to completion
  - collect output
  - submit next job
- Processor cycles very expensive at the time
- Jobs involved reading, writing data to/from tapes
- Cycles were being spent waiting for the tape!

Timesharing Operating Systems

- Solution
  - store multiple batch jobs in memory at once
  - when one is waiting for the tape, run the other one
- Basic idea of timesharing systems
- Fairness primary goal of timesharing schedulers
  - let no one process consume all the resources
  - make sure every process gets “equal” running time
- Preemptive multitasking:
  - give OS power to interrupt process
  - programmer freed from scheduling
  - scheduler can enforce fairness

Real-Time Operating Systems

- ***Def:*** *A real-time operating system is an operating system that supports the construction of real-time systems*
- Main goal of an RTOS scheduler: meeting deadlines
- If you have five homework assignments and only one is due in an hour, you work on that one
- Fairness does not help you meet deadlines

Real-Time Operating Systems

The following are the three key requirements

1. **The timing behavior of the OS must be predictable.**
   ∀ services of the OS: Upper bound on the execution time!
   An RTOS must be deterministic:
   - unlike standard Java,
   - short times during which interrupts are disabled,
   - contiguous files to avoid unpredictable head movements.
2. **OS must manage the timing and scheduling**
   - OS possibly has to be aware of task deadlines; (unless scheduling is done off-line).
   - OS must provide precise time services with high resolution.
3. **The OS must be fast**
   Practically important.

Real-Time Operating Systems

attributes	RTOS	conventional OS
virtual memory	none	yes
application prog.	single purpose	general purpose
scheduling policy	based on priority	fairness
I/O processing	by app. prog.	through drivers
delivery of massive data between tasks	speed up by passing pointers, etc.	use system service like pipe, mailbox
resource allocation	static	dynamic (runtime)
file system or disk	none or limited	yes

Real-Time Operating Systems

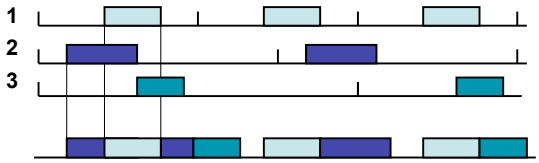
- The basic guiding concept for most operating systems and schedulers is to strive for good average performance while in real time systems meeting the deadlines is far more important than average performance.
- **Protected memory:** while important for the stability of the system it carries a high overhead in a real time environment because of the added cost to a context switch and the caches that need to be flushed each context switch (the TLB for example).
- **User and kernel modes:** while protecting the systems from user processes adds significant overhead because each mode switch incurs a performance penalty because of the trap instructions and cache flushes.
- **Memory paging:** can increase the cost of a context switch by several magnitudes if the memory pages we need were switched out. This causes a lot of unpredictability in the timing.
- **Dynamic memory allocation:** this is a critical feature used by most modern application, however it creates memory fragmentation and as a result adds to timing unpredictability.

Priority-Based Scheduler

- Typical RTOS based on fixed-priority preemptive scheduler
- Assign each process a priority
- At any time, scheduler runs highest priority process ready to run
- Process runs to completion unless preempted

Priority-Based Preemptive Scheduling

- Always run the highest-priority runnable process

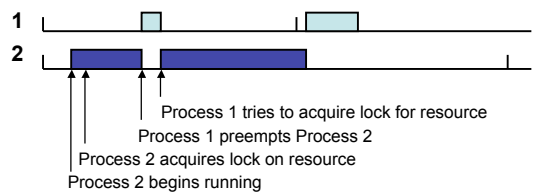


Priority-Based Preemptive Scheduling

- Multiple processes at the same priority level?
- A few solutions
  - simply prohibit: Each process has unique priority
  - time-slice processes at the same priority
    - extra context-switch overhead
    - no starvation dangers at that level
  - processes at the same priority never preempt the other
    - more efficient
    - still meets deadlines if possible

Priority Inversion

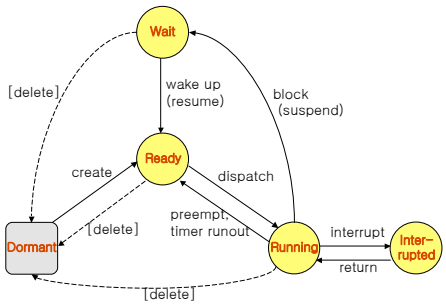
- RMS and EDF assume no process interaction
- Often a gross oversimplification
- Consider the following scenario:



Tasks

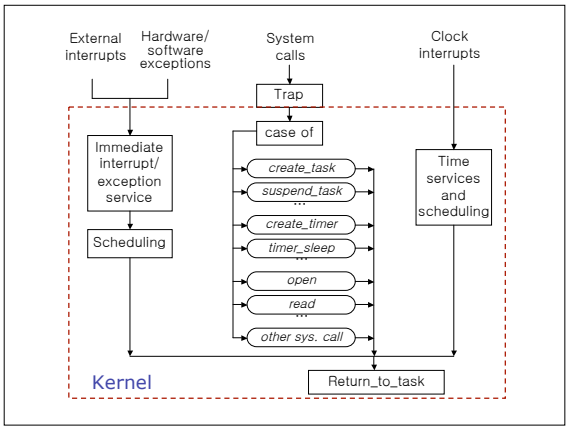
- Tasks
  - tasks and task control block (TCB)
    - task ID
    - starting address
    - task context (PC, SP, registers, etc)
    - task parameters
    - scheduling information
    - synchronization information
    - timer information
    - other information
  - task parameters for real-time OS
    - task type
    - phase
    - period
    - relative deadline
    - priority
    - release time
    - loss rates

Tasks



Real-Time Tasks

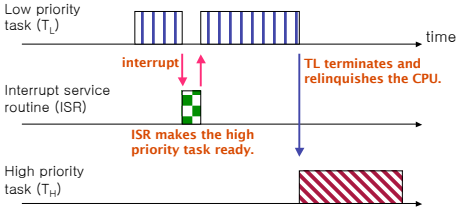
- Task scheduling
  - inserted into the schedule queue when ready
  - scheduling policies, scheduling disciplines
    - non-real-time
      - FCFS, round robin, priority-based, shortest job first, etc.
    - real-time
      - rate monotonic, earliest deadline first, etc.
  - priority assignment
    - criteria
      - importance(criticality), period, deadline
      - execution time, arrival time, elapsed time after ready
    - method
      - dynamic (at runtime)
      - static (at initialization)



Kernel

- Kernel
  - performs basic functions for task management and intertask communication
  - usually, resident in main memory
- Non-preemptive kernel
  - once executed on CPU, a task continues executing until it releases control
  - in case of interrupt, it resumes execution upon completion of ISR
  - advantages
    - completion time of task execution is predictable (if execution time of ISR is negligible)
    - no dynamic interference of data between tasks
  - disadvantages
    - high priority tasks may be blocked unnecessarily
    - schedulability test difficult
    - low responsiveness

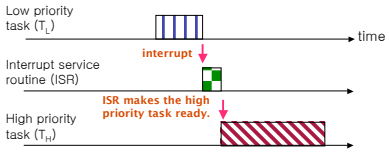
Non-Preemptive Kernel



RTOS

- Preemptive kernel
  - upon transition to ready state, a high priority task will be dispatched by suspending lower priority task in execution
  - in case of interrupt, a higher priority task will be scheduled upon completion of ISR
  - advantage
    - higher priority task: predictable completion time – real-time
    - responsiveness
  - disadvantage
    - conflict in using shared data and resources

Preemptive Kernel



Synchronization

- Mutual exclusion
  - shared data
    - resides in the same address space
    - e.g.: global variables, pointers, buffers, lists
    - access through critical sections
  - inter-task communication using shared data
    - concurrent accesses must be controlled to avoid corrupting the shared data → mutual exclusion
  - mutual exclusion techniques
    - disabling interrupts
    - test-and-set operation
    - disabling the scheduler
    - semaphores

Memory Usage

- programs (executable codes)
- task control block, TCB
- stack space