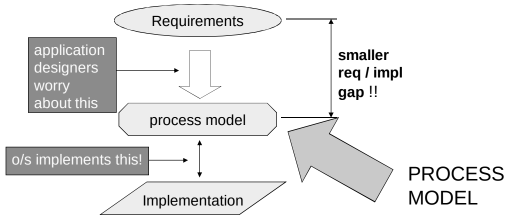
SYSC 5701
Operating System Methods for Real-Time Applications

Event-Driven Process Model

Winter 2014

RECALL: Motivation reduced requirements/implementation gap



Process Model

- an abstract model for concurrent systems design, which provides:
  - appropriate blend of sequential (simple) and event-driven (realistic) mindsets
  - concurrency framework for identifying and describing concurrent activities
  - mechanisms for concurrent interaction
  - mechanisms to ensure that high-priority work is not delayed by low-priority work

Abstraction of Platform

- want platform **semi**-independence, ignore:
  - machine-level details where possible
    - e.g. processor register use
  - implementation details of process model
- BUT …maintain necessary links to machine
  - e.g. h/w interrupts
    I/O h/w
    exception mechanisms

Must Be Practical !

- must have practical & understandable execution expectations !!

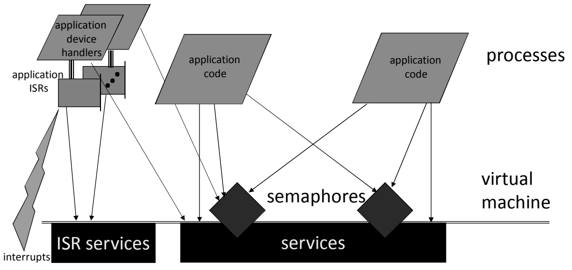
  if practical implementation not possible

  then resulting models must be redesigned for implementation

- Is an understand implementation an issue?
  - Will people ever look at the underlying implementation?

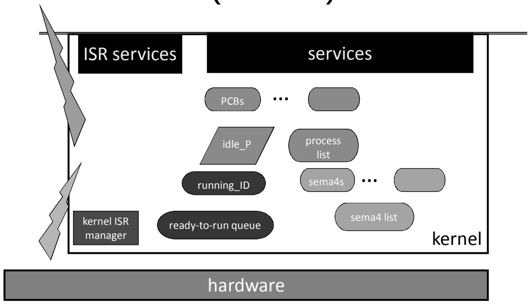
Ideally, we want to get to this: … slide 50+

Process Model Abstraction



And …

Process Model Implementation (Kernel)



Process

- basic unit of concurrency in process model
  - Contain sequential program code
- processes may **execute concurrently**
  - Concurrency: physical and/or apparent
- a <u>semi</u>-autonomous program fragment
  - process interactions (IPC)
- identifiable as an artifact in both the design and implementation
- FreeRTOS → tasks

Periodic Processes in Theory vs. Practice?

- Process vs. Liu's tasks?
- Process vs. Liu's jobs?

- Concern:
  - How is periodic release managed?
    - By process model or by code design?

      → later!

Process Control State

- each process has a control state, e.g.:
  - **running** currently executing
  - **blocked** not eligible to run

- NOTE: list of possible states will grow during implementation discussions!

InterProcess Communication (IPC)

- IPC mechanisms are part of process model
  - Managed by the kernel
- allows processes to interact
  - synchronize : e.g. semaphore
  - communicate : e.g. message-passing
- can processes share memory space?
  - lightweight → yes
  - heavyweight → no
    - heavyweight typically needs MMU h/w

Lightweight Process (Thread) vs. Heavyweight Process

- differences will be discussed later
- do not get tripped up (bogged down) in concern over differences at this point

- lets assume <u>lightweight</u> processes for now

  → can share memory space

  → can share variables

  →FreeRTOS tasks are lightweight

Semaphore

- An IPC object in the process model
- used for
  - mutual exclusion : programmed control over access to shared resources
    - e.g. to avoid interference
  - synchronization : coordinate progress
    - e.g. consumer waits for producer
- FreeRTOS has semaphores

Semaphore Concept

- abstract synchronization gate
- process requests permission to pass gate
- either: allowed to pass the gate (continue executing) or blocked at the gate until permission is granted later
- multiple concurrent requests to pass gate are serialized by the semaphore
  - only one at a time through gate

Operational Model of Semaphore

- internal resources:
  - protected **counter**
    - initialized to some non-negative value
      - default or specified at creation
  - **blocked_Q** : process queue – initially empty
- current value of counter = number of processes that may pass gate before gate closed
- counter = 0 → gate closed!
- blocked processes "wait" in blocked_Q

yields processor!

no busy waiting!

Semaphore Operations

- **wait** and **signal**
- wait : request permission to pass gate
- signal : allow one more process to pass gate
- operations share counter and blocked Q
- must be interference free !
  - process model implementation (Kernel) must serialize wait & signal code and protect internal data structures

Wait Operation

```
Wait        // request permission to pass gate
//  this is serialized (protected) code!
if   counter > 0
        then  // gate is open – so pass
              decrement counter
              // decrement may close the gate!
        else  // gate is closed
              block process (pause execution) and
              enqueue  process in blocked_Q
```

Signal Operation

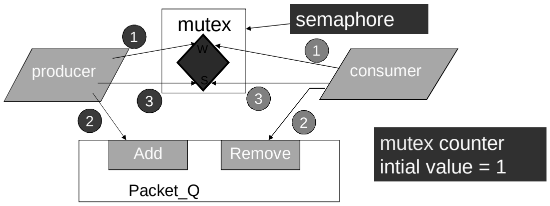
```
Signal      // allow one more process to pass gate
   //  this is serialized (protected) code!
   if   blocked_Q  is empty
        then  // no processes are waiting to pass
              increment counter
              // allows a future process to pass
        else  // at least one process waiting
              dequeue process from blocked_Q and
              resume execution of the (unblocked) process
```

Protected Semaphore Operations

- Kernel implements internal protection
- Application developers do not have to worry about how this is done
- Makes implementing protected application code easier!
  - Protect using semaphores
  - Reduce the development gap!

Mutual Exclusion (mutex)

- recall Stream-2-Pipe example: want mutually exclusive access to **packet_Q**



semaphore

mutex counter
intial value = 1

Adding to Packet_Q

```
  Protected_Add(  P :  packet_buffer )
1 {  mutex.Wait;          // gain exclusive access
2    Packet_Q.Add( P );       // add to Q
3    mutex.Signal;     // release exclusive access
  }
```

- Application code uses mutex semaphore to protect access to shared Packet_Q
  - Packet_Add is protected! (serialized access)

Removing from Packet_Q

```
  Protected_Remove( var P :  packet_buffer )
1 {  mutex.Wait;          // gain exclusive access
2    Packet_Q.Remove( P );     // remove from Q
3    mutex.Signal;      // release exclusive access
  }
```

- Application code uses mutex semaphore to protect access to shared Packet_Q
  - Packet_Remove is protected! (serialized access)

Synchronization

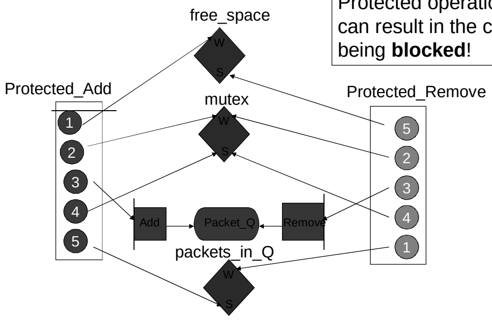
- recall Stream-2-Pipe – only want to allow:
  Remove : only when a packet is available
  Add : only when there is space for the packet

- need more semaphores to synchronize!
  - will introduce 2 more semaphores …

Additional Semaphores

packets_in_Q : semaphore = **0**;
// used to block Removers until a packet ready
// initially no packets ready count = 0
// gate is closed!

free_space : semaphore = **Q_Size**;
// used to block Adders until space is available
// initially all spaces in Packet_Q are available
// count = Q_Size … gate is open!



NB: calling the Protected operations can result in the caller being **blocked**!

Revised Add to Packet_Q

```
Protected_Add( P : packet_buffer )
1 { free_space.Wait; // get space in Packet_Q
2   mutex.Wait;          // gain exclusive access
3   Packet_Q.Add( P );     // add to Q
4   mutex.Signal;        // release exclusive access
5   packets_in_Q.Signal;   // packet now ready!
  }
```

Protected_Add and **Packet_Q.Add(P);** are set in bold.

Revised Remove From Packet_Q

```
Protected_Remove( var P : packet_buffer )
1 { packets_in_Q.Wait;        // wait for packet
2   mutex.Wait;             // gain exclusive access
3   Packet_Q.Remove( P );   // remove from Q
4   mutex.Signal;       // release exclusive access
5   free_space.Signal;   // one more freed space!
  }
```

Common Synchronization Bug **DEADLOCK**!

- suppose Remove as above, but this Add :

```
Protected_Add( P : packet_buffer )
{ mutex.Wait;              // gain exclusive access
  free_space.Wait;       // get space in Packet_Q
  Packet_Q.Add( P );   // add to Q
  packets_in_Q.Signal;         // packet now ready!
  mutex.Signal;            // release exclusive access
}
```

can a process be blocked in a protected region? ☹

Add/Remove Deadlock Scenario

- suppose Packet_Q is full
- producer has another packet → Add
  1. mutex.Wait; → passes
  2. free_space.Wait; → blocked!
- now consumer tries to remove
  1. packets_in_Q.Wait; → passes
  2. mutex.Wait; → blocked!

DEADLOCK!

Process Model Implementation Kernel (a.k.a. Nucleus)

- run-time support for process model
- reduces req/impl gap
- typically: small, efficient, fast
- often highly configurable
  - operating environment & functionality
- central core of an “operating system” for a real-time embedded system

Basic Kernel Functionality

- **process management** services:
  - **scheduling** of processes to processor(s)
  - **context switching**: block a process, remove it from processor(s) and install new process on processor(s)
- **IPC** services (e.g. semaphores)
- may provide **additional services** (configurable?)
  - e.g. resource management such as process-related memory management (MMU)

Kernel Services Impl'n

- services re-entrant and internally protected

  (re-entrant vs. recursive ?)
- invoke services using **software interrupt**
  - (a.k.a. trap, supervisor call)
  - similar behaviour to hardware interrupt
    - save state, transfer to Kernel ISR
    - can change processor protection mode
  - flexible run-time vs. link-time resolution

    ↑ ↑

    dynamic vs. static

Kernel's View of a Process

- each process requires memory resources:
  - executable code – read-only – can be shared
  - local data variables – read / write – not shared
  - stack – each process must have own stack!
    - separate "threads of control"
- processes can share global variables and I/O resources
  - **share with care**!!!
  - heap ?? heap manager ??

Kernel Keeps Information About Each Process

- **process id** – to uniquely identify the process
- **current logical state** (running, blocked, etc.)
  - needed for scheduling decisions
- **allocated resources** – memory, I/O devices, o/s resources (e.g. semaphores)
  - needed for process management
- **processor execution state** – register values
  - needed for context switch
- **priority** – needed for scheduling

Process Control Block: **PCB**

- **data record** (e.g. struct) used by kernel to manage info relevant to one process
  - each process has a corresponding PCB
- fields for relevant process info
- may also include link fields used to manage PCB in various dynamic lists maintained by kernel

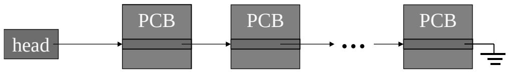
Process ID

- to identify and refer to process at run-time
  - IDs must be unique
  - want to use ID to gain efficient access to PCBs
  - cheap solution:

    ID = pointer to process' PCB

Process List

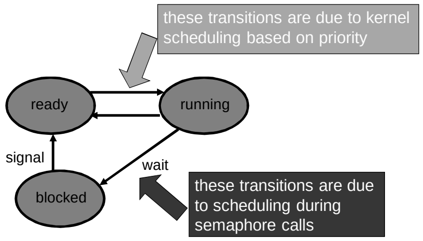
- list of PCB's of all processes that currently exist
  - often: list uses PCB pointers
- often implemented using a single head pointer variable and a "next" field in each PCB
  - PCBs in a linked list



Process State

- state transitions are due to kernel scheduling
- running and blocked are no longer sufficient
  - what if running, but not on a processor?
  - (i.e. waiting for a turn on a processor)
- introduce **ready** state – eligible to run, but not currently on a processor

Process State Transitions



these transitions are due to kernel scheduling based on priority

these transitions are due to scheduling during semaphore calls

Ready Processes

- kernel maintains **ready-to-run queue**
  → **RTR**
- queue (using PCB pointers) of ready processes
- need:
  - head pointer variable in kernel
  - field in each PCB for linking into RTR queue
    - just like field for linking into process list

Running Processes

- kernel maintains a **running_P** variable
  - uniprocessor: ID of currently running process
    - often just use PCB at head of RTR queue
  - multiprocessor: multiple running process IDs
    - can't use (single) head of RTR queue
      → one running_P variable per processor

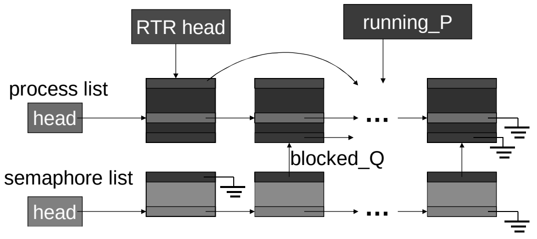
Semaphore Management

- semaphore control block for each semaphore
  - **count**
  - **blocked_Q**
- semaphore runtime ID
  → pointer to control block
- kernel maintains sema4_list
  → list of all semaphore control blocks

Blocked Processes

- how to implement blocked_Q ?
- one possible solution:
  - semaphore control block contains blocked_Q head pointer
  - each PCB contains a field for linking into appropriate blocked_Q
  - Link all processes blocked on semaphore into a list

Multiple Lists and Queues



Process Creation

to set up environment for process, need to know:

- **stack requirements** (for stack creation)
  - alternative: default size
    - → let process create bigger stack if needed
      - BUT difficult to delete process and recover used memory if stack is not known to kernel
- **static data memory required**?
- **execution start address**
- **priority**
- set up PCB → save process creation info

Process Initialization

- process initial state? ready?
  - system initialization concerns! (more later!)
- ensure process queued in appropriate queue(s)
  - process list
  - ready-to-run? other? depends on state?

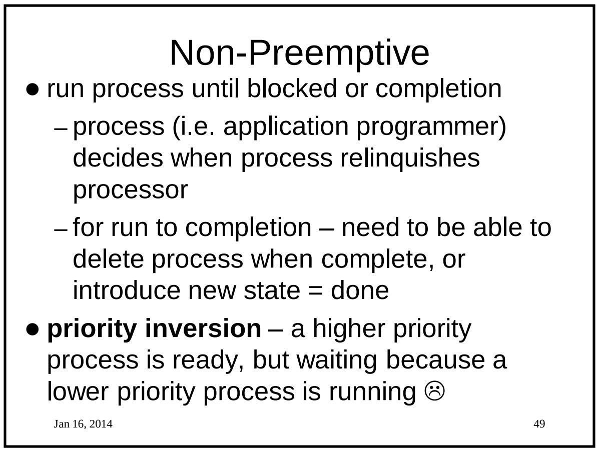
- process deletion is more complex – later!

Process State Modification

- these events may require the kernel to change the state of a process:
  - running process finishes scheduled work
  - running process calls wait or signal
  - an interrupt results in another process becoming ready
    - e.g. an I/O interrupt that releases an I/O related process

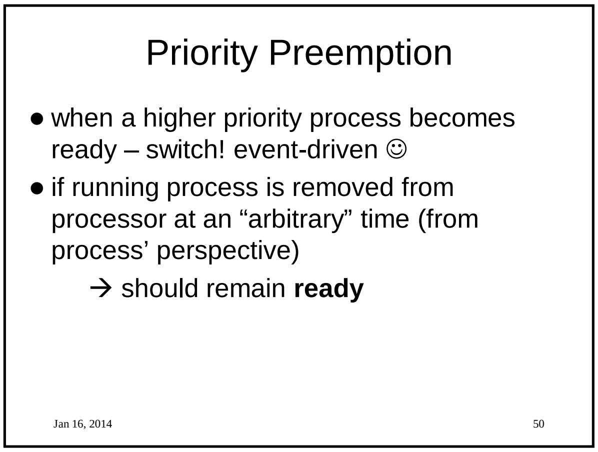
Scheduling Points

- when a process changes state, kernel must make a scheduling decision (dynamic!)
- has the state change resulted in a situation where a context switch should be performed?
- if yes → do a context switch
- if no → leave current process running



Non-Preemptive

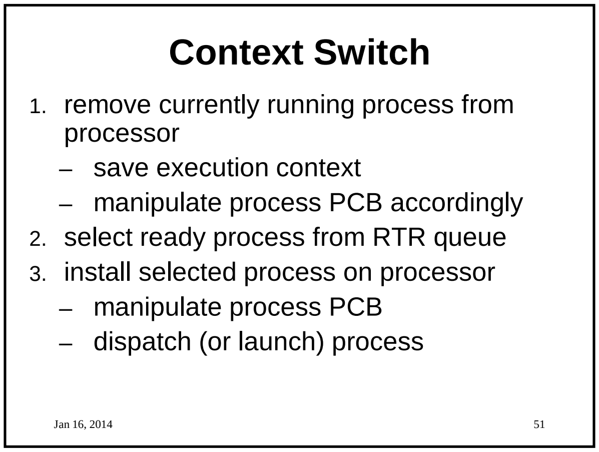
- run process until blocked or completion
  - process (i.e. application programmer) decides when process relinquishes processor
  - for run to completion – need to be able to delete process when complete, or introduce new state = done
- **priority inversion** – a higher priority process is ready, but waiting because a lower priority process is running ☹



Priority Preemption

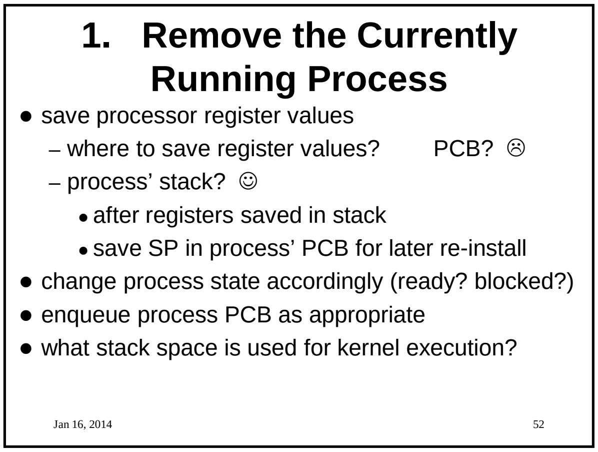
- when a higher priority process becomes ready – switch! event-driven ☺
- if running process is removed from processor at an "arbitrary" time (from process' perspective)

  → should remain **ready**



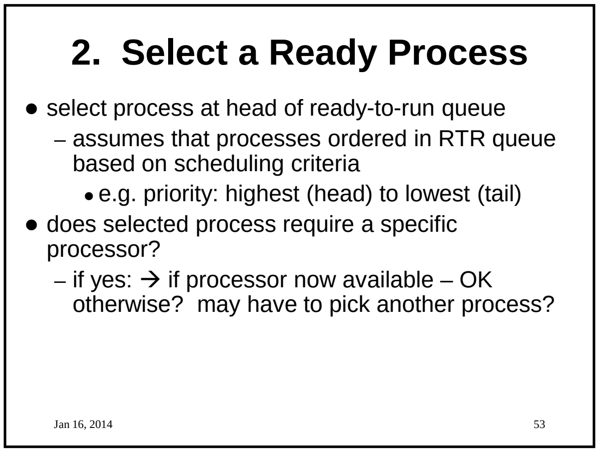
Context Switch

1. remove currently running process from processor
   - save execution context
   - manipulate process PCB accordingly
2. select ready process from RTR queue
3. install selected process on processor
   - manipulate process PCB
   - dispatch (or launch) process



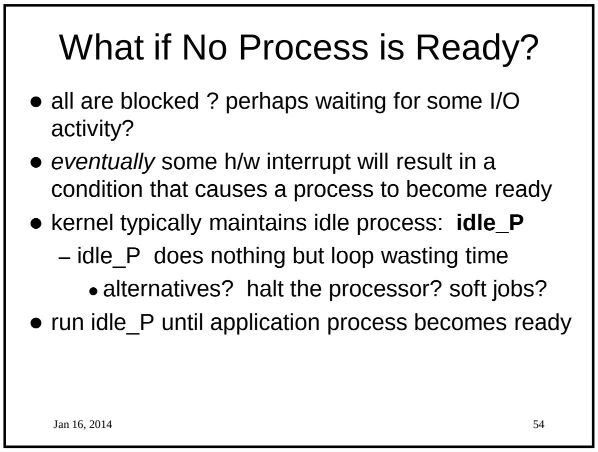
1. Remove the Currently Running Process

- save processor register values
  - where to save register values? PCB? ☹
  - process' stack? ☺
    - after registers saved in stack
    - save SP in process' PCB for later re-install
- change process state accordingly (ready? blocked?)
- enqueue process PCB as appropriate
- what stack space is used for kernel execution?



2. Select a Ready Process

- select process at head of ready-to-run queue
  - assumes that processes ordered in RTR queue based on scheduling criteria
    - e.g. priority: highest (head) to lowest (tail)
- does selected process require a specific processor?
  - if yes: → if processor now available – OK otherwise? may have to pick another process?



What if No Process is Ready?

- all are blocked ? perhaps waiting for some I/O activity?
- *eventually* some h/w interrupt will result in a condition that causes a process to become ready
- kernel typically maintains idle process: **idle_P**
  - idle_P does nothing but loop wasting time
    - alternatives? halt the processor? soft jobs?
- run idle_P until application process becomes ready

3. Install Selected Process on Processor

- dequeue process from RTR
- record process ID in running_P
- change process state to running
- get stack pointer (SP) from process' PCB
  - restore saved registers
- once PSW and IP are restored → launched process is executing!
- **NB**: **MUST** release any internal kernel protection before PSW and IP are restored

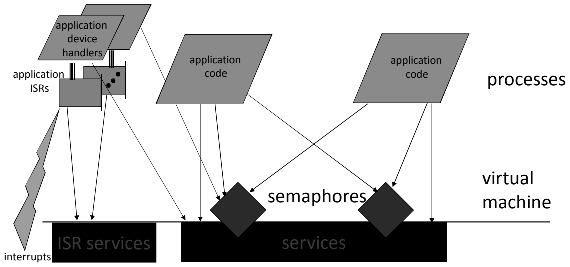
H/W Interrupt Events

- kernel provides (at least initial) handling of h/w interrupts
- device handlers are typically implemented as processes above the kernel
- device handler priority is a design issue – often priority is higher than application processes

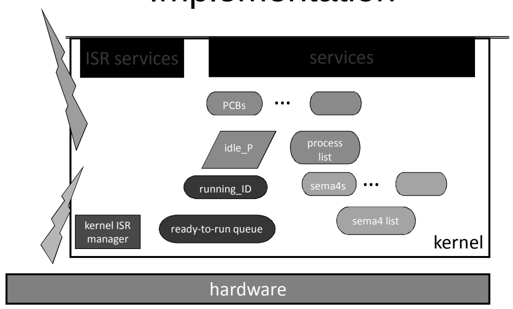
Kernel Services for H/W Interrupts

- application supplied h/w interrupt service routine (ISR) associated with (**bound**-to) a h/w interrupt
- **special** IPC functionality to allow ISRs to interact with processes (e.g. device handlers)
  - kernel code takes advantage of assumptions associated with h/w ISRs
  - not handled the same way as process invoked IPC requests
    - optimize speed and efficiency

Process Model Abstraction



Process Model Implementation

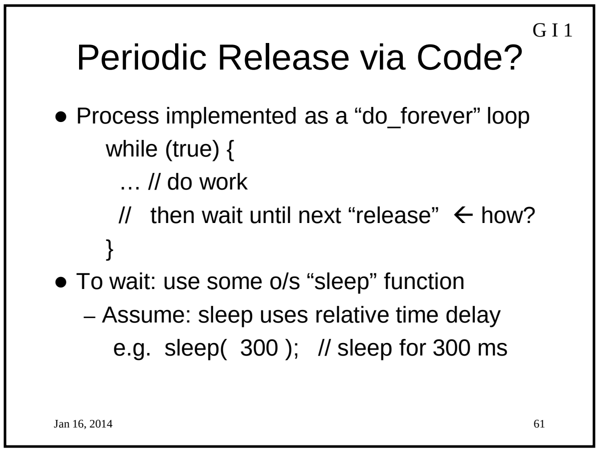


Some "Gnarly" Issues

Often, no easy answers!

gnarl•y *Pronunciation:* när'l**E**
adj., gnarl•i•er, gnarl•i•est.
Slang. **distasteful; distressing; offensive**;

1. periodic process release
2. memory for kernel managed objects
3. system initialization
4. dynamic removal of kernel managed objects
5. exception handling



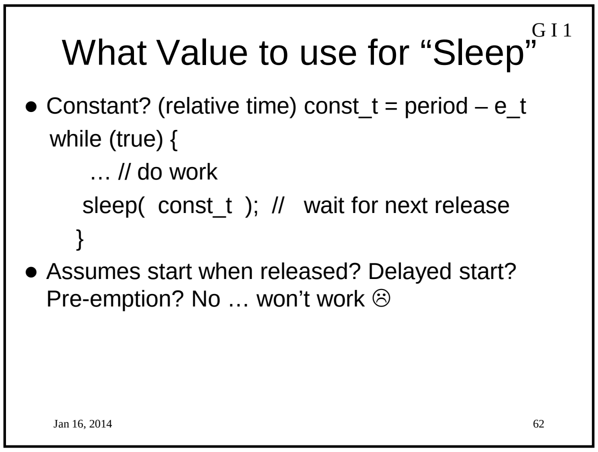
G I 1

Periodic Release via Code?

- Process implemented as a “do_forever” loop

```
while (true) {
  … // do work
  //   then wait until next “release”  ← how?
}
```

- To wait: use some o/s “sleep” function
  - Assume: sleep uses relative time delay
    e.g. sleep(300);   // sleep for 300 ms



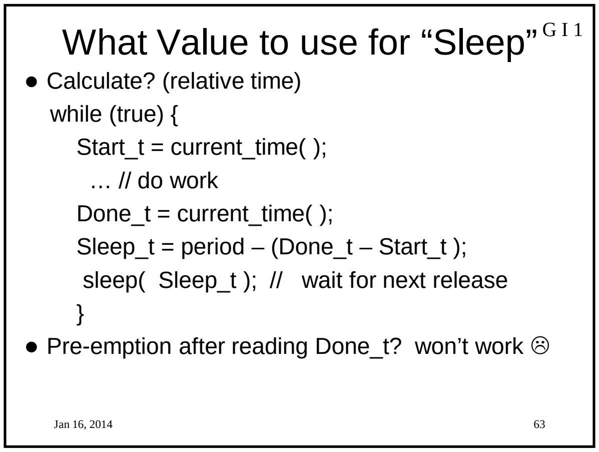
G I 1

What Value to use for “Sleep”

- Constant? (relative time) const_t = period – e_t

```
while (true) {
     … // do work
    sleep(  const_t  );  //   wait for next release
   }
```

- Assumes start when released? Delayed start? Pre-emption? No … won’t work ☹



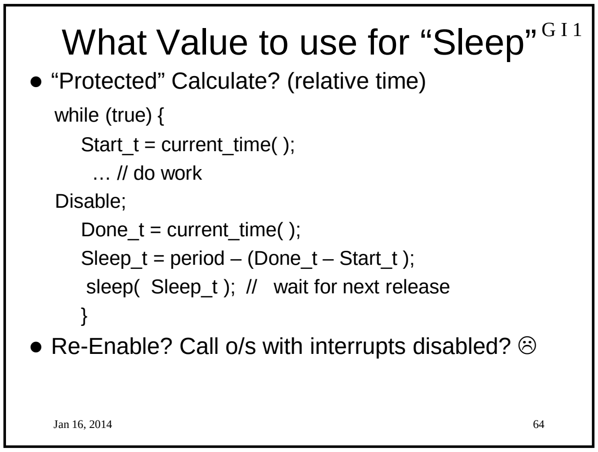
G I 1

What Value to use for “Sleep”

- Calculate? (relative time)

```
while (true) {
    Start_t = current_time( );
      … // do work
    Done_t = current_time( );
    Sleep_t = period – (Done_t – Start_t );
     sleep(  Sleep_t );  //   wait for next release
    }
```

- Pre-emption after reading Done_t? won’t work ☹



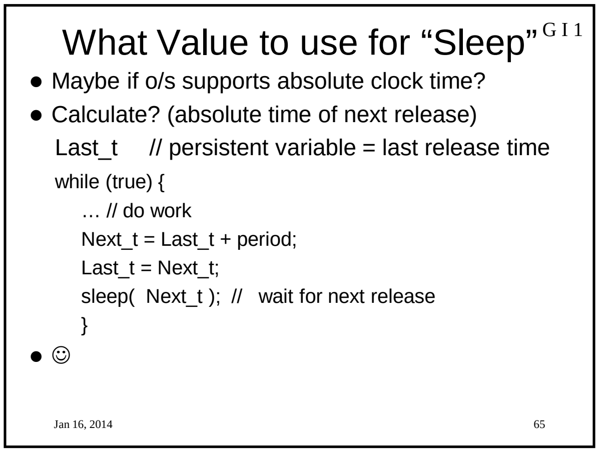
G I 1

What Value to use for “Sleep”

- “Protected” Calculate? (relative time)

```
while (true) {
    Start_t = current_time( );
      … // do work
Disable;
    Done_t = current_time( );
    Sleep_t = period – (Done_t – Start_t );
     sleep(  Sleep_t );  //   wait for next release
    }
```

- Re-Enable? Call o/s with interrupts disabled? ☹



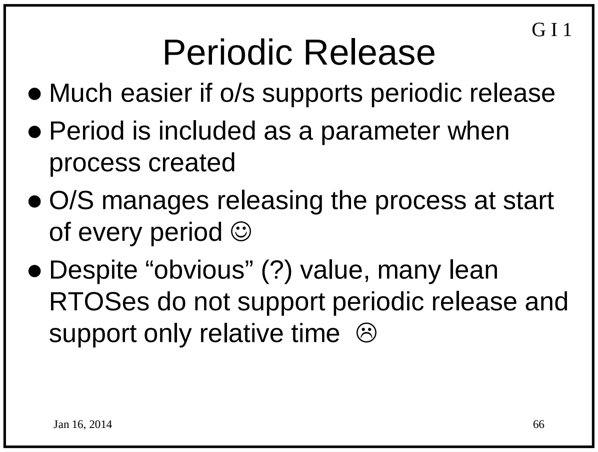
G I 1

What Value to use for “Sleep”

- Maybe if o/s supports absolute clock time?
- Calculate? (absolute time of next release)

```
Last_t     // persistent variable = last release time
while (true) {
    … // do work
    Next_t = Last_t + period;
    Last_t = Next_t;
    sleep(  Next_t );  //   wait for next release
    }
```

- ☺



G I 1

Periodic Release

- Much easier if o/s supports periodic release
- Period is included as a parameter when process created
- O/S manages releasing the process at start of every period ☺
- Despite “obvious” (?) value, many lean RTOSes do not support periodic release and support only relative time ☹

G I 2

Memory for Kernel’s Use

- dynamically? (from where?)
- memory manager module?
  - part of o/s? part of kernel?
  - part of language support code?
  - part of application code?
- is manager initialized before kernel needs it?
- what should kernel do if no memory available? → exception?

G I 2

Obtaining Memory (more)

- pre-allocate statically?
  - fixed number of system objects?
  - simple vs. limitations!
- shift responsibility to application ☺
  - when app calls kernel to create object must pass pointer to block of memory to be used by kernel to manage object

G I 2

Application Supplies Memory

```
e.g.  sema4  create_sema4 (
            initial_value: integer ;
            sema4_control_block : pointer )
```

- returns runtime ID of created sema4 object
  - pointer? trouble!
    - → application code has access to block!

G I 3

System Initialization

- in real-time, embedded applications → o/s & application code often linked into single load module (distributed system? load module**s**?)
- what executes first? application vs. o/s?
  - o/s must init before o/s can provide services
- default application init code? (“main”)
- high-level language-relevant init code too!?
  - language’s run-time support code?

G I 3

O/S Inititialization

- might include:
  - initialize internal structures
  - setup vectors for service calls
  - timer h/w and ISR
  - other h/w? – e.g. memory manager?
  - create idle process

G I 3

Initial Creation of Processes and Semaphores

- process initial state? = ready?
- could a process run before other required processes and semaphores have been created? ☹
- careful attention to order of object creation
  - ensure not possible for a process to be created before objects necessary for interaction have been created
  - cyclic dependencies?
  - can be complex – hard to modify/evolve

G I 3

Initialization Mode?

- o/s does not dispatch any application processes until “go” call made to change mode to “normal”
- application init code creates objects needed, then calls “go” to release created processes
- system complexity too? multiprocessor? network?

G I 4

Dynamic Process Removal

- why delete a process? done vs. abort
- ran to completion – nothing more to do (done)
  - typically “safe” – application tidies up first
- application termination of activity – application no longer wishes to perform related work (abort)
  - e.g. “cancel” button pressed
- recovering from exception – delete, then restart subset of system (abort)
- terminating system in a controlled manner (abort)

G I 4

Why Might Abort-Deletion Be Difficult?

- process might currently be using resources
  - in a critical section? release of mutex sema4?
- manipulating state-dependent h/w device?
  - preempt h/w access?
  - leave h/w in unexpected state?
- other processes might be expecting participation
- will deletion upset cooperation patterns?

G I 4

What About Objects Created By the Process?

- delete these too?
- memory allocation?
- recall sema4 management blocks example
  - dangling references to objects?

G I 4

Permission to Delete a Process?

- arbitrary?
- process can delete itself (terminate on completion)
- parent/child process creation tree
  - parent: creates child processes
  - process can only be deleted by a direct ancestor
  - root of tree – can delete any process
- kernel vs. application?
- exception handlers?

G I 5

Exception Handling

- (should be) major concern in real-time systems
- what to do if something goes wrong?
- fault tolerance? – recover and continue
- reliability?
- hard to find solid discussions in generic texts!

Examples of Exception Conditions

- could be due to application or o/s or h/w (or combinations)
- deadlock – application flaw?
- divide by zero
- stack overflow
  - unexpected bursts of events
  - stack use by ISRs?

More Exception Conditions

- memory protection fault
  - accessing a dangling reference?
- hardware errors
  - e.g. network communication failure
- too many events to process and still meet timing constraints
  - event bursts, h/w failures

Sensing Exception Conditions

- redundant s/w checks – e.g. CRC checks
- compiler inserts test code – performance?
  - compilation switches
- h/w senses – interrupts
- timed services: “watchdog” timer

E.G Timed Sema4.Wait Call

- specify maximum time process can be blocked
  - fixed maximum or parameter?
- if process blocked for specified time → timeout
  - exception?
  - kernel releases process?
    - need return-code to indicate normal vs. timeout return from service call

Kernel Support of Timed Wait

- kernel handles timer ISR – “tick”
  - duration? configuration parameter?
- kernel might maintain some notion of a “clock”
  - accumulated ticks ?
  - time-of-day ?
- PCB has timeout field
  - unit resolution?
  - ticks-until-timeout count vs. clock time
  - clock time: absolute vs. relative time ?

Timed Wait (con’t)

- periodically (depends on resolution of timeout units) kernel extends behaviour of timer ISR to handle service timing
  - release processes if necessary
  - overhead !

More Timeout Issues

- how to manage return-code?
  - – ready + return-code field in PCB?
- priority of timed-out processes?
- what if application wants to use timer interrupt?
  - – daisy-chain after kernel’s use?
    - → jitter

What to do When an Exception Occurs?

- log details – how? accessible if system crashes?
- fix (if possible) and continue – ignore failure
  - – e.g. I/O error: reset h/w device
  - – hope protocols recover?
- re-attempt failed work
  - – preempt relevant processes
    - roll back to a point before exception ?
      - → capability to rollback = overhead!
  - – try again

More on What to Do

- re-attempt is often built into soft systems
  - – e.g. communication protocols
- continue with reduced capability
  - – restore capabilities when system repaired
- admin/operator interface to system

- crash and burn ☹

Processing Exceptions

- functionality?
  - – application-specific ☺
  - – kernel: generic 😐
- attach application-specific handlers to kernel?

An Observation

(Pearce and others)

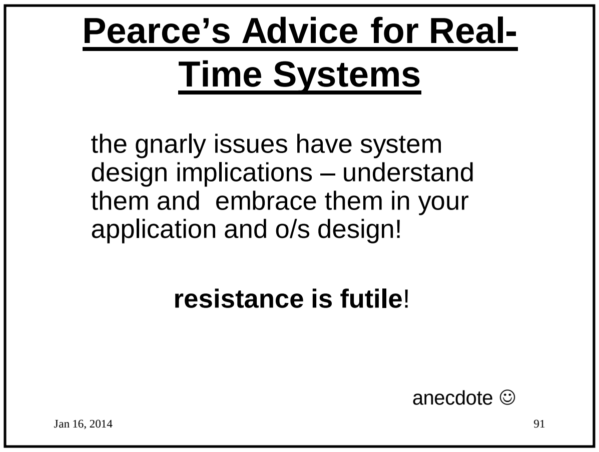
- exception handling in real-time applications adheres to Pareto Distribution: 20 / 80 split
  - – 20 % code → “normal” (80%) behaviour
  - – 80 % code → exception processing (20%)
    - – tricky!
- what were you trained to develop?

Grinding a Software Engineering Axe

theoreticians often argue that design should be abstract & implementation independent
→ nice in theory, *but*

… **in practice** …

real-time system implementation quirks associated with specific process model details and gnarly issues inevitably influence design decisions!



Pearce's Advice for Real-Time Systems

the gnarly issues have system design implications – understand them and embrace them in your application and o/s design!

resistance is futile!

anecdote ☺

Jan 16, 2014 91