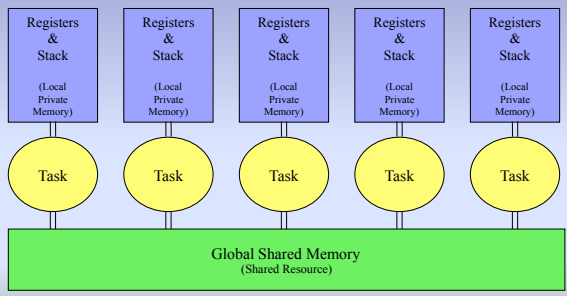
One more thing...

- **YAK kernel functions must be *reentrant***
- What does this mean, and why is it important?
- Let's revisit the shared-data problem
  - We saw the problem between ISRs and task code
  - New, but not a surprise: the problem arises **between RTOS tasks**
  - Tasks often share data and helper functions, and inconsistency can occur if shared data is accessed non-atomically
- Let's revisit how memory is used with an RTOS

Task memory organization



Key point: all non-stack memory is shared

Global data

- Tasks, ISRs, and the kernel share *all* global data
- Easy, convenient for task to send data to another task using global variables
- Shared data problems can arise when
  - tasks access global variables defined by **application** code, and
  - kernel functions access global variables defined by **kernel**
- Embedded developers must be aware of *where* C variables are stored
  - Critical to identify those that can cause problems

Example: Figure 6.6
What can go wrong here?

```
struct {
    long lTankLevel;
    long lTimeUpdated;
} tankdata[MAX_TANKS];

void vRespondToButton(void)
{ /* high priority task */
  int j;
  while (TRUE) {
    !! Block until button pressed
    j = !! ID of button pressed
    !! output lTankLevel[j]
    !! output lTimeUpdated[j]
  }
}
```

```
void vCalculateTankLevels(void)
{ /* low priority task */
  int i = 0;
  while (TRUE) {
    !! read float levels in tank i
    !! do lots of calculations
    /* store result */
    tankdata[i].lTimeUpdated = !! current time
    tankdata[i].lTankLevel = !! current level
    !! pick next tank to handle, etc.
  }
}
```

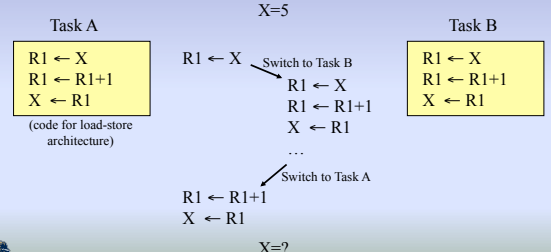

A more subtle problem: Figure 6.7
What can go wrong here?

```
void Task1(void)
{
  ⋮
  vCountErrors(9);
  ⋮
}

void Task2(void)
{
  ⋮
  vCountErrors(11);
  ⋮
}
```

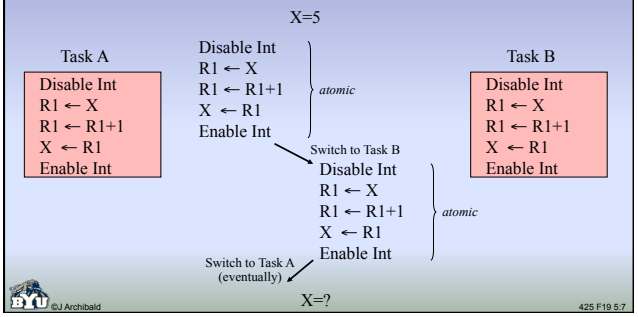
```
static int cErrors;

void vCountErrors(int cNewErrors)
{
  cErrors += cNewErrors;
}
```


Shared data: the problem revisited

Suppose both Task A and Task B increment shared variable X



Shared data: a solution



Reentrant functions

- The original code in vCountErrors() is not reentrant
- A function is reentrant if it can be called by multiple tasks and still work correctly in all cases
  - Regardless of timing of interrupts or task switches
- Kernel functions can be called by multiple tasks and ISRs
  - For correct operation, your YAK kernel functions must be reentrant

Reentrant: definition (Wikipedia)

A computer program or routine is described as reentrant if it can be safely executed concurrently; that is, the routine can be re-entered while it is already running. To be reentrant, a function

- must hold no static data,
- must not return a pointer to static data,
- must work only on the data provided to it by the caller,
- must not rely on locks to singleton resources, and
- must not call non-reentrant functions.

Reentrant: definition (our text)

To be reentrant, a function must not

1. use variables in a non-atomic way, unless they are local to the calling task, or
2. call any other functions that are not reentrant, or
3. use the hardware in a non-atomic way.

Shared variables

- Does requirement that functions be reentrant mean they cannot access shared variables?
  - No, shared variable access is okay if atomic
- Atomicity seldom occurs naturally. Programmer must
  - recognize critical sections in code (portion that must be atomic to work correctly), then
  - make each critical section atomic by disabling/enabling interrupts (or using alternate approaches, usually with more overhead).

Shared hardware

- Possible hardware problems with non-reentrant code:
  - Garbled output on printer or screen
  - Garbled transmission over wireless link
- Interleaved use of hardware by tasks is problematic
  - Related problems will not arise if code is reentrant
- Reentrant code requirement:
  - Use of hardware must be atomic
  - Code must finish each hardware "transaction" that it starts before something else can use the hardware

What's okay in reentrant code?

```
static int static_int;
int public_int;
int initialized = 4;
char *string = "Where does this string go?";
void *vPointer;

void function (int parm, int *parm_ptr)
{
  static int static_local;
  int local;
  .
  .
  .
}
```

Suppose function accesses all variables listed on this slide.

Is it reentrant?

What's okay in reentrant code?

```
static int static_int;
int public_int;
int initialized = 4;
char *string = "Where does this string go?";
void *vPointer;

void function (int parm, int *parm_ptr)
{
  static int static_local;
  int local;
  .
  .
  .
}
```

Definitely a problem: accesses to global variables

What's okay in reentrant code?

```
static int static_int;
int public_int;
int initialized = 4;
char *string = "Where does this string go?";
void *vPointer;

void function (int parm, int *parm_ptr)
{
  static int static_local;
  int local;
  .
  .
  .
}
```

Definitely a problem: accesses to global variables

Definitely not a problem: local variables

What's okay in reentrant code?

```
static int static_int;
int public_int;
int initialized = 4;
char *string = "Where does this string go?";
void *vPointer;

void function (int parm, int *parm_ptr)
{
  static int static_local;
  int local;
  .
  .
  .
}
```

Definitely a problem: accesses to global variables

Possibly a problem: local copy of pointer

Definitely not a problem: local variables

Reentrancy: the bottom line

- To determine if a function is reentrant, you must examine all variables it accesses
  - You must know where each variable is stored
  - Non-atomic accesses to variables not on task stack are a problem
  - Example: in initial version of clib.s, output from print was sometimes messed up
    - Function was not reentrant; print used global char array to generate string
    - Code fix: a local array (allocated on stack) was used instead
- In embedded systems, the implementation details matter

Applying the three rules:
Is display() reentrant?

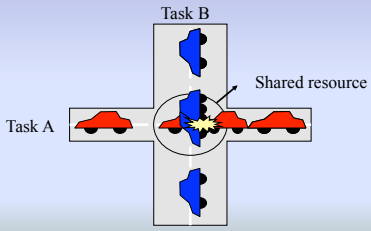
```
BOOL fError;   /* flag set by something else */
void display (int j)
{
  if (! fError)
  {
    printf("Value: %d", j);
    j = 0;
    fError = TRUE;
  }
  else
  {
    printf("Could not display value");
    fError = FALSE;
  }
}
```

1a. Does it use global variables?
  b. Does it use them in a non-atomic way?
2. Does it call non-reentrant functions?
3. Does it use hardware non-atomically?

Subtle cases

- What if the only global variable access is increment?
  - Is access to x for x++ necessarily atomic?
    - Likely to be on 8086 – unless operand size is 32-bits!
    - Will not be on many embedded platforms
  - Best practice: use approach that works for all target platforms
    - Little downside to adding *short* critical sections
- What if only access to global variable is a read?
  - Is access to x for y = x necessarily atomic?
    - Not for 16-bit value on 8-bit architecture, etc.

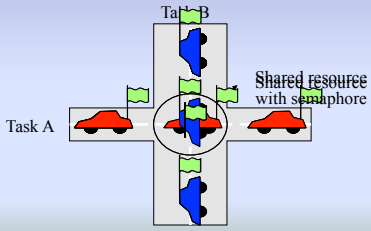
Shared resources



Shared resources

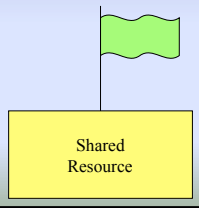
- When two or more tasks try to use a **shared resource** at the same time, failures can occur.
- To avoid failure, tasks must ensure **mutual exclusion**: when one task is using the shared resource, other tasks are excluded.
- **Critical section**: section of code in which a shared resource is used non-atomically.
- **Mutual exclusion**: only one task is allowed in critical section at a time.

Shared resources



Semaphore

- One definition:
  - A semaphore is an object that limits shared access to another object



Semaphore

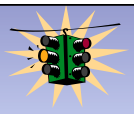
- Other definitions:
  - An apparatus for conveying information by means of visual signals, as a light whose position may be changed
  - Any of various devices for signaling by changing the position of a light, flag, etc.
  - A system of signaling, esp. a system by which a special flag is held in each hand and various positions of the arms indicate specific letters, numbers, etc.

Examples?

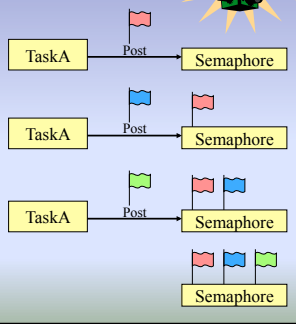
Vocabulary

- Types of semaphores:
  - Binary (Boolean): simplest
  - Counting (integer): original type, used in YAK
- Assorted names for semaphore operations:
  - *Get* and *Give*
  - *Take* and *Release*
    - Terminology used in the book
  - *Pend* and *Post*
    - Terminology used in this class and in YAK
  - *P* and *V*
    - From the Dutch: *Proberen* (test) and *Verhogen* (raise)
    - Terminology used by Edsger Dijkstra who invented semaphores

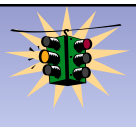
Semaphores

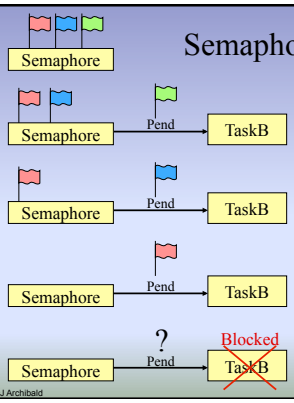


Post Semaphore
(or Release Semaphore)
Semaphore is incremented.



Semaphores





Pend Semaphore
(or Take Semaphore)
Semaphore is decremented.

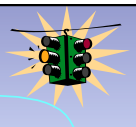
If result is negative, task is blocked.

Using counting semaphores

- How is a semaphore with values > 1 useful?
  - Example: represents the number of available resources:
    - Memory blocks, processors, I/O channels, etc.
- What is meaning if value < 0?
  - Absolute value is number of waiting tasks
- Counting semaphores easily simulate binary semaphores:
  - Context will insure that they're not incremented past 1
    - Initialize to 1, increment only when resource released, etc.
    - May be decremented below zero: two states are {1} and {<1}

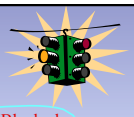
Using semaphores

Mutual Exclusion
Using a semaphore to solve the shared resource problem



Using semaphores

Signaling (Synchronization)
Using a semaphore to signal another task



Semaphore usage: Fig. 6.12

```
struct {
  long lTankLevel;
  long lTimeUpdated;
} tankdata[MAX_TANKS];

void vRespondToButton(void)
{ /* high priority task */
  int i;
  while (TRUE) {
    !! Block until button pressed
    i = !! ID of button pressed
    TakeSemaphore( );
    !! output tank level, timestamp
    ReleaseSemaphore( );
  }
}
```

```
void vCalculateTankLevels(void)
{ /* low priority task */
  int i = 0;
  while (TRUE) {
    !! read float levels in tank i
    !! do bunches of calculations
    TakeSemaphore( );
    !! set tankdata[i].lTimeUpdated
    !! set tankdata[i].lTankLevel
    ReleaseSemaphore( );
    !! pick next tank to handle, etc.
  }
}
```


Discussion

- If TakeSemaphore( ) is called and the semaphore is not available, when does control return to the task?
  - The task is blocked until the semaphore is available
  - The unblocked task runs when it is the highest priority ready task
  - Note the role of the RTOS in supporting this functionality
- Can a high priority task be forced to wait for a lower priority task?
  - Is this a good thing?
  - Is it avoidable?
- What if a task is interrupted while it holds a semaphore?
  - Can a context switch cause problems for other tasks?
  - More on this shortly…

Semaphores vs. disabling interrupts

- What are advantages of each approach?
- How does each affect response time of system?
- Which is easiest to use?
- Which is more bug-prone?

- Observations:
  - Disabling interrupts has less overhead but affects entire system
  - Semaphore is more targeted, but has more runtime overhead
  - Disabling interrupts more easily understood

Semaphore usage: Fig. 6.14

```
#define PRIORITY_READ 11
#define PRIORITY_CONTROL 12
#define STK_SIZE 1024
static unsigned RStk[STK_SIZE];
static unsigned CStk[STK_SIZE];
static int iTemperatures[2];
OS_EVENT *p_semTemp;

void main (void)
{
  OSInit( );
  OSTaskCreate (vRdTmpTsk, NULLP,  (void *)
    &RStk[STK_SIZE], PRIORITY_READ);
  OSTaskCreate (vCtrlTask, NULLP, (void *)
    &CStk[STK_SIZE], PRIORITY_CONTROL);
  OSStart( );
}
```

```
void vRdTmpTsk (void) {
  while (TRUE) {
    OSTimeDly (5);
    OSSemPend (p_semTemp, WAIT_FOREVER);
    !! read in iTemperatures[0];
    !! read in iTemperatures[1];
    OSSemPost (p_semTemp);
  }
}

void vCtrlTsk(void) {
  p_semTemp = OSSemCreate(1);
  while (TRUE) {
    OSSemPend (p_semTemp, WAIT_FOREVER);
    if (iTemperatures[0] != iTemperatures[1])
      !! set off howling alarm;
    OSSemPost (p_semTemp);
    !! do other useful work
  }
}
```


Discussion and questions

- Operational notes:
  - OS prefix indicates kernel functions (in µC/OS)
  - Overall functionality similar to YAK
  - Semaphore pointer initialized, then used with pend and post
  - OS_EVENT struct represents semaphore
- Questions:
  - What is WAIT_FOREVER parameter in pend?
  - How was semaphore created?
    - What does initialization value of 1 mean?
  - **Can anything go wrong with this code?**

The problem

- Can't guarantee that semaphore will be initialized before first pend call that uses it. (Why?)

- "**If you write embedded code that relies on this kind of thing, you will chase mysterious bugs for the rest of your career.**"

- Best practice?
  - Put semaphore initialization call in main, where it is guaranteed to execute before call to OSStart, and therefore before any task runs.

Using semaphores to make functions reentrant

- Surround critical region within function by calls to pend and post
- Function is then reentrant: can safely be called by any task at any time

```
calling_function( )
{
    Non-critical Region
    reentrant_function();
    Non-critical Region
}
```

```
reentrant_function( )
{
    SemPend();
    Critical Region
    SemPost();
}
```


Designing with semaphores

- How many semaphores should you use in your application?
- Consider extreme cases and the tradeoffs involved:
  - Single semaphore to protect *all* shared resources:
    - Simple, easy to keep track of
    - Tasks accessing different resources can block each other
  - Many semaphores, one for each shared resource:
    - With many semaphores, increased likelihood of coding errors, confusion
    - A little more storage and processing overhead for system
    - Minimum amount of blocking, best response time
    - Lower priority task won't cause higher priority task to block unless both try to acquire the same resource

Using semaphores

- How does the RTOS know which semaphore protects which data?
  - Actually, it has no knowledge of this
  - RTOS does not know what semaphore protects, so it cannot enforce program-specific rules of semaphore usage
- Correct usage is responsibility of application programmer:
  - Must create each semaphore, initialize to proper value, and use consistently throughout code
- Semaphores are only as good as the application programmer using them

Semaphores for signaling

```
static char a_chPrint[10][21]; /* output buffer */
static int iLinesTotal;
static int iLinesPrinted;
static OS_EVENT *semPrinter;

void vPrinterInterrupt (void)
{
  if (iLinesPrinted == iLinesTotal)
     /* report done: release semaphore */
     OSSemPost(semPrinter);
  else
     /* report not done: print next line */
     vHardwarePrinterOutputLine(
        a_chPrint[iLinesPrinted++]);
}
```

Task formats output, prints first line.
Printer interrupts after each line prints.
ISR prints remaining lines, signals task.

```
void vPrinterTask(void)
{
  BYTE byError;
  int wMsg;
  semPrinter = OSSemInit(0);
  while (TRUE)
  {
    wMsg = (int) OSQPend (QPrinterTask,
       WAIT_FOREVER, &byError);
    !! Format the report into a_chPrint
    iLinesTotal = !! count of lines in report
    /* print first line of report */
    iLinesPrinted = 0;
    vHardwarePrinterOutputLine(
       a_chPrint[iLinesPrinted++]);
    OSSemPend(semPrinter, WAIT_FOREVER,
       &byError);
  }
}
```


Discussion

- In code example, semaphore was used to signal
  - Sending task does a post
  - Receiving task does a pend
  - Compare with mutual exclusion: same task calls pend, then post
- Initial semaphore value must be chosen carefully
  - In example, semaphore set to 0 (not available); typical for signaling
- Task pends on both queue and semaphore
  - Is order interchangeable?
  - Order of multiple pends is potential cause of bugs

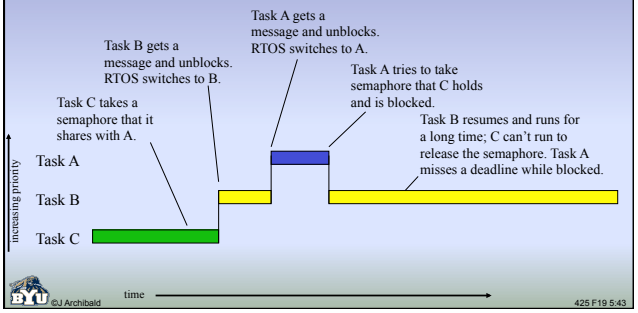
Potential errors with semaphores

- Forgetting to take (pend) the semaphore
- Forgetting to release (post) the semaphore
- Taking or releasing the wrong semaphore
- Initializing a semaphore to the wrong value
- Holding the semaphore too long
- Priority inversion
- Deadlock, or deadly embrace

} to be discussed shortly

"**Every use of semaphores is a bug waiting to happen.**" (page 167)

Priority inversion: the problem



Discussion

- Is this a problem that arises in real systems?
- Why is it called *priority inversion*?
  - RTOS scheduling choice reflects static task priority, not true dynamic importance of tasks
  - Most important task after Task A blocks is really lower priority task holding resource A is waiting for

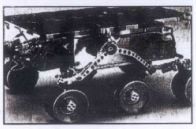
Priority inversion: solutions

- Can the higher priority task simply take the desired resource from the lower priority task?
  - Would violate atomicity requirement in critical sections; not good!
- Can priority of lower priority task be boosted temporarily?
  - **Priority inheritance**: RTOS temporarily raises priority of task holding resource to that of higher priority waiting task
  - How might this work in previous example?
    - RTOS could *swap* priorities of tasks A and C until C releases semaphore, then switch priorities back to original values
    - While C holds semaphore and A is blocked, C could be preempted only by tasks with higher priority than A. (B won't run.)

Bug of the Month

Man Finds Bugs on Mars

Wherever a computer goes, bugs are sure to follow. When the Mars Pathfinder developed a glitch, NASA had to somehow upload new code without losing valuable time needed for exploration. The most confounding bug on the Pathfinder mission appeared July 10. Steven Stolper, software engineer for the Mars Pathfinder, calls it "one in a million, insidious, and hard to replicate." The snafu arose because the OS, Wind River's VxWorks, developed a mutual-exclusion problem: A low-priority function (in this case, recording weather) interfered with the system's multitasking schedule. The system couldn't finish all the tasks it needed to, missed a real-time deadline, and then shut itself down. "It's a kind of interplanetary Control-Alt-Delete," says Stolper. "When things go wrong, the system goes into a power-safe mode and waits for ground control to help out." Without a fix being implemented, this problem would replay itself over and over.



Pathfinder bugs inhibited the Rover.

To identify the bug, engineers recreated the malfunction on Earth, identified the offending subroutine, and uploaded the binary difference between the new code and the buggy code on the Pathfinder. –Jason Krause

Send yours to jkrause@mgh.com!

OCTOBER 1997 BYTE 23

Mars rover bug was priority inversion

- The commercial RTOS (VxWorks) includes versions of pend functions with and without priority inheritance
- Programmer intended to use pend with priority inheritance, but used other pend by mistake
- Lengthy, medium priority task to record weather caused a (blocked) high-priority task to miss a deadline
- Code detected missed deadline, assumed major malfunction, shut down the rover, waited for instructions from earth

"Every use of semaphores is a bug waiting to happen."


```
int a;
int b;
AMXID hSemaphoreA;
AMXID hSemaphoreB;
void vTask1 (void)
{
   ajsmrsv (hSemaphoreA, 0, 0);
   ajsmrsv (hSemaphoreB, 0, 0);
   a = b;
   ajsmris (hSemaphoreB);
   ajsmris (hSemaphoreA);
}

void vTask2 (void)
{
   ajsmrsv (hSemaphoreB, 0, 0);
   ajsmrsv (hSemaphoreA, 0, 0);
   b = a;
   ajsmris (hSemaphoreA);
   ajsmris (hSemaphoreB);
}
```

Another potential problem

- Relevant details:
  - 2 global variables, each protected by a semaphore
  - Both semaphores must be obtained before operations involving both variables
- What can go wrong here?
  - Circular dependence, or **deadlock**
- Problem is seldom this obvious
  - Usually hidden within nested function calls; resources acquired at different points in call sequence

Deadlock

Definition: a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does
- Also called deadly embrace

Deadlock can occur only if four conditions are met:
1. Mutual exclusion: a resource is either available or assigned to a task
2. Hold and wait: tasks already holding a resource may request new resources
3. No preemptive stealing: only a task holding a resource may release it
4. Circular wait: two or more tasks form a circular chain where each process waits for a resource that the next process in the chain holds

("Coffman conditions", 1971)

Legislated deadlock

When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.

Satirical summary of law said to have been passed by a state legislature under the control of the "Know-Nothing" party

Livelock

- Similar to deadlock in that no progress is made
  - Key difference: state of the tasks involved constantly change with respect to each other, none progressing
- Real-world approximation:
  - Two people meet in narrow corridor, and both move to same side to let the other pass; both repeatedly switch to the other side, but they do so at the same time – both are stuck
- Conceptually, could result from trying to avoid deadlock:
  - Could allow tasks to detect that second desired semaphore is held by another task, so they release the first and start again
  - Conceptually possible to stay in sync so neither makes progress

Starvation

- Occurs when a task is perpetually denied necessary resources such as CPU time or memory
  - The task can never do what it needs to do
- What is responsible for avoiding CPU starvation with conventional OS?
  - The OS scheduler should ensure that each process gets its turn
- What is responsible for avoiding CPU starvation with RTOS?
  - Application code
  - What can developer change if one or more tasks are starved?

Discussion

- In RTOS setting
  - What is state of tasks involved in deadlock?
  - What is state of tasks involved in livelock?
  - What is state of task that is CPU starved?

Semaphores: odds and ends

- How many values can a semaphore have?
  - Binary semaphores: 2 values (1 or 0)
  - Counting semaphores: more than 2
- Some RTOSs support a "mutex semaphore"
  - Automatically addresses priority inversion problem
  - Terminology not consistent across all RTOSs
- If multiple tasks are waiting for a semaphore, which gets unblocked when it is released? What options exist?
  - Highest priority task (YAK)
  - Longest waiting task
  - Some RTOSs let you choose between these two

"Every use of semaphores is a bug waiting to happen."

Protecting shared data

- Three distinct alternatives (not counting "programming tricks"):
  - Disabling interrupts
    - Most drastic way to protect data
    - Affects response time of all tasks and ISRs
    - Fast: generally takes just a single instruction to disable/enable
    - Only method that works when sharing data with an ISR
  - Taking semaphores
    - Affects only those tasks that take the same semaphore
    - No impact on tasks that don't take the semaphore, or on ISRs
    - Creates new possibilities for programming errors
  - Disabling task switches (a.k.a. locking the scheduler)
    - Lock function simply sets global flag, and unlock clears it
    - Scheduler won't switch tasks if flag set
    - Response time: affects all tasks, but not ISRs
    - Available in many RTOSs (not in YAK, but would be easy to add)

Calling restrictions on kernel functions

- ISRs must never call kernel routines that might block caller
  - Examples: delay, pend
- But ISRs *are* allowed to call post functions
  - Assumes that post functions never block the caller
- Tasks have no comparable restrictions on functions they can call
- Thought experiment:
  - What would happen if ISR did pend on semaphore in our kernels?
  - Key insight: pend code assumes that it has been called by a task

Implementing YKSemPend

```
void YKSemPend(semaphore)
{
    !! disable interrupts
    if (semaphore.value-- > 0)
    {
        !! enable interrupts
        return;
    }
    !! block calling task: take its TCB out of the ready list
    !! modify TCB, put in pending list
    !! call scheduler
    !! enable interrupts
}
```


Implementing YKSemPost

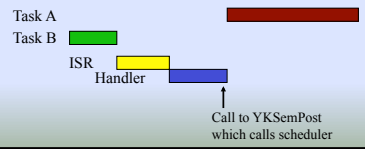
```
void YKSemPost(semaphore)
{
    !! disable interrupts
    if (semaphore.value++ >= 0)
    {
        !! enable interrupts
        return;
    }
    !! find TCB of highest priority task waiting for this semaphore
    !! modify TCB of that task, place in ready list
    !! call scheduler (if not in ISR)
    !! enable interrupts
}
```


Semaphore implementation notes

- Runtime overhead should be low on pend when semaphore available, and on post when no other tasks waiting
  - Overhead in other cases depends on how ready + blocked lists implemented
- Initial value of semaphore is set by YKSemCreate
  - Must be chosen carefully
- Why not call scheduler in YKPost if called in an ISR?
  - More on this shortly (next slide)
  - Why not enable interrupts before calling scheduler?

Calling the scheduler

- Consider this scenario:
  - Task B interrupted, handler calls YKSemPost
  - YKSemPost calls scheduler at end
  - Scheduler decides to run Task A, calls dispatcher
- Is this a problem?

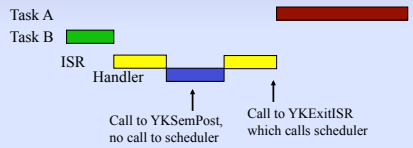


Calling the scheduler

- Consider this scenario:
  - Task B interrupted, handler calls YKSemPost
  - YKSemPost calls scheduler at end
  - Scheduler decides to run Task A, calls dispatcher
- Is this a problem?
  - Yes! ISR has not finished: EOI command not executed, nesting level count not updated
- Solution?
  - In YAK, do not call scheduler from YKSemPost if in ISR
  - Applies to all post routines
- In scenario on previous slide, when should scheduler be called?

Calling the scheduler

- What should happen:



Preemption in YAK

- Which of these YAK kernel functions must include a call to the scheduler in their source code?
  - **YKNewTask**
  - **YKDelayTask**
  - **YKInitialize**
  - **YKRun**
  - **YKEnterMutex**
  - **YKExitMutex**
  - **YKEnterISR**
  - **YKExitISR**
  - **YKScheduler**
  - **YKDispatcher**
  - **YKTickHandler**
  - **YKSemCreate**
  - **YKSemPend**
  - **YKSemPost**
  - **YKQCreate**
  - **YKQPend**
  - **YKQPost**

Reentrancy in YAK

- Which of these YAK kernel functions must be reentrant?
  - **YKNewTask**
  - **YKDelayTask**
  - **YKInitialize**
  - **YKRun**
  - **YKEnterMutex**
  - **YKExitMutex**
  - **YKEnterISR**
  - **YKExitISR**
  - **YKScheduler**
  - **YKDispatcher**
  - **YKTickHandler**
  - **YKSemCreate**
  - **YKSemPend**
  - **YKSemPost**
  - **YKQCreate**
  - **YKQPend**
  - **YKQPost**

Reentrancy in YAK

- What functions in YAK application code must be reentrant?
  - main()?
  - Each task function?
  - Each ISR?
  - Each interrupt handler?
  - Helper functions called by tasks?

Calling restrictions in YAK

- For YAK functions below, is direct call from ISRs/handlers a concern?

Pick correct response for each function:
1. Not a problem
2. Poor design choice, but should not crash system
3. Will cause malfunction

  - **YKNewTask**
  - **YKDelayTask**
  - **YKInitialize**
  - **YKRun**
  - **YKEnterMutex**
  - **YKExitMutex**
  - **YKEnterISR**
  - **YKExitISR**
  - **YKScheduler**
  - **YKDispatcher**
  - **YKTickHandler**
  - **YKSemCreate**
  - **YKSemPend**
  - **YKSemPost**
  - **YKQCreate**
  - **YKQPend**
  - **YKQPost**

Midterm #1

- In class, closed book, no calculators.
  - 1 page front and back
  - Some true/false, short answer, multiple choice, circle all correct, etc.
- Recommendations: how to prepare
  - Study midterm #1 and solution from F18 on class webpage
  - Review text, paying careful attention to definitions
  - Review slides on material not directly covered in text
  - Review YAK specs, application code from labs
  - Review C topics (from HW, class discussion)
  - Review your design notes and code for your kernel
  - Review case study (Athens Affair)
  - Study list of review topics on class webpage

Midterm #1 review topics

- Book and lecture topics
  - design issues in embedded real-time systems
  - critical performance issues in embedded real-time systems
  - memory address space conventions, RAM and ROM
  - watchdog timers
  - interrupt mechanisms, hardware and software
  - saving and restoring context
  - the shared data problem
  - atomicity and critical sections
  - interrupt latency
  - alternatives to disabling interrupts
  - software architectures: alternatives and tradeoffs
    - round robin or polled-loop architectures
    - round robin with interrupts
    - function-queue scheduling
    - real-time operating system

Midterm #1 review topics

- Book and lecture topics (continued)
  - tasks, task states and transitions, multitasking
  - typical RTOS functions
    - creating tasks, semaphores, etc.
    - deleting tasks
    - delaying tasks
    - changing task priority
    - inter-task communication and synchronization
  - RTOS data structures: task control blocks (TCBs)
  - scheduling, alternative scheduling policies
  - priorities among tasks, ISR relative to tasks
  - preemption, how it is accomplished
  - shared data problem, reentrant code
  - C variable storage
  - semaphores, typical functionality, potential problems
  - semaphores: signaling vs. protecting shared data
  - semaphore variants

Midterm #1 review topics

- Book and lecture topics (continued)
  - deadlock, livelock, starvation
  - priority inversion, priority inheritance
  - context switching, saving, restoring
  - role and functionality of dispatcher
  - challenges of designing and debugging real-time system code
- Lab and HW basics
  - essential C topics
  - implementation of C constructs in assembly
  - stacks, stack frames, conventions in compiled C code
  - make files
  - ISR essentials
  - YAK functions and conventions
  - saving, restoring context in YAK

Midterm #1 review topics

- Lab and HW basics (continued)
  - 8086-based tools (compiler, assembler): general usage
  - emu86 simulator: usage, debugging features
  - conventions in compiled C: stack frames, parameters, return values
  - 8086 instructions, operations
    - enabling, disabling interrupts (IMR and flag register)
    - CPU actions on interrupt, iret
    - interrupt jump table
    - supporting nested interrupts
- Additional reading: case study
  - Operational details of underlying technology
  - Facts of specific case study (what happened and why important)
  - Implications (technological, social, political, etc.)

Exam issues: clarity

- Be precise in your language on essay and short answer questions
  - Can a task "interrupt" another task?
  - Can a task "block" another task?
  - Is task function ever "called"?
  - Is ISR ever "called"?
  - Is it "deadlock" if a (possibly buggy) task holds a semaphore too long or never gives it up?
  - Is "code" interchangeable with "global variables"?
    - Example: "Don't use code non-atomically."
  - Are variables "controlled" by semaphores?
- Careful writing is always important

Problems from ends of chapters

- Recommendation:
  - Study those from Chapters 4-6
- Let's look at some examples

Problem 6.2

Is this function reentrant?

```
int strlen (char *p_sz)
{
  int iLength;
  iLength = 0;
  while (*p_sz != '\0')
  {
    ++iLength;
    ++p_sz;
  }
  return iLength;
}
```


Problem 6.3

Which of the numbered lines in the function would lead you to conclude that this function is not reentrant?

```
static int iCount;

void vNotReentrant (int x, int *p)
{
    int y;

    y = x * 2;                  /* line 1 */
    ++p;                        /* line 2 */
    *p = 123;                   /* line 3 */
    iCount += 234;              /* line 4 */
    printf("New count: %d", x); /* line 5 */
}
```


Problem 6.6

- For each of the following situations, discuss which of the three shared-data protection mechanisms seems most likely to be best and explain why.
  - Task M and task N share an **int** array, and each must often update many elements in the array.
  - Task P shares a single **char** variable with one of the interrupt routines.

Problem 6.8

Assume that this code is the only code in the system that uses the variable iSharedDeviceXData. The routine vGetDataFromDeviceX is an interrupt routine. Now suppose that instead of disabling all interrupts in vTaskZ, as shown, we disable only the device X interrupt, allowing all other interrupts. Will this still protect the iSharedDeviceXData variable? If not, why not? If so, what are the advantages (if any) and disadvantages (if any) of doing this compared to disabling all interrupts?

```
int iSharedDeviceXData;

void interrupt vGetDataFromDeviceX (void)
{
        iSharedDeviceXData =
            !! Get data from device X hardware
        !! reset hardware
}

void vTaskZ (void)
{
        int iTemp;
        while (FOREVER)
        {
            /* other code here */
            !! disable interrupts
            iTemp = iSharedDeviceXData;
            !! enable interrupts
            !! compute with iTemp
        }
}
```


Problem 6.10

Consider this statement: "In a nonpreemptive RTOS, tasks cannot 'interrupt' one another; therefore there are no data-sharing problems among tasks."

Do you agree?

Problem 5.1

Consider a system that controls traffic lights at a major intersection. It reads from sensors that notice the presence of cars and pedestrians, it has a timer, and it turns the lights red and green appropriately. What [software] architecture might you use for such a system? Why? What other information, if any, might influence your decision?

Problem 4.1: Does this approach avoid a shared data problem?

```
static int iSeconds, iMinutes, iHours;

void interrupt vUpdateTime (void)
{
    ++iSeconds;
    if (iSeconds >= 60)
    {
        iSeconds = 0;
        ++iMinutes;
        if (iMinutes >= 60)
        {
            iMinutes = 0;
            ++iHours;
            if (iHours >= 24)
                iHours = 0;
        }
    }
    !! Deal with HW
}
```

```
void vSetTimeZone (int iZoneOld, int iZoneNew)
{
    int iHoursTemp;

    /* Get current hours */
    disable();
    iHoursTemp = iHours;
    enable();

    !! adjust iHoursTemp for new time zone
    !! adjust for daylight savings time also

    /* save the new hours value */
    disable();
    iHours = iHoursTemp;
    enable();
}
```

Code based on Figure 4.17

Problem 4.2: The code below has a shared data bug.

```
static long int lSecondsToday;

void interrupt vUpdateTime (void)
{
    ...
    ++lSecondsToday;
    if (lSecondsToday == 60 * 60 * 24)
        lSecondsToday = 0L;
    ...
}

long lSecondsSinceMidnight(void)
{
    return (lSecondsToday);
}
```

(a) How far off can return value of function be if sizeof(long) is 32 and word size is 16 bits?

(b) How far off can return value of function be if sizeof(long) is 32 and word size is 8 bits?

Problem 4.3: What additional bug lurks in this code, even if registers are 32 bits in length?

```
static long int lSecondsToday;

void interrupt vUpdateTime (void)
{
    ...
    ++lSecondsToday;
    if (lSecondsToday == 60 * 60 * 24)
        lSecondsToday = 0L;
    ...
}

long lSecondsSinceMidnight(void)
{
    return (lSecondsToday);
}
```

What can happen if system has another interrupt that is higher priority than timer interrupt for vUpdateTime and that calls lSecondsSinceMidnight?


```
static int iTemperaturesA[2], iTemperaturesB[2];
static BOOL fTaskCodeUsingTempsB = FALSE;
void interrupt vReadTemperatures (void) {
    if (fTaskCodeUsingTempsB)
    {
        iTemperaturesA[0] = !! read in value from HW
        iTemperaturesA[1] = !! read in value from HW
    }
    else
    {
        iTemperaturesB[0] = !! read in value from HW
        iTemperaturesB[1] = !! read in value from HW
    }
}

void main (void) {
    while (TRUE)
    {
        if (fTaskCodeUsingTempsB)
            if (iTemperaturesB[0] != iTemperaturesB[1])
                !! Set off howling alarm;
        else
            if (iTemperaturesA[0] != iTemperaturesA[1])
                !! Set off howling alarm;
        fTaskCodeUsingTempsB = !fTaskCodeUsingTempsB;
    }
}
```

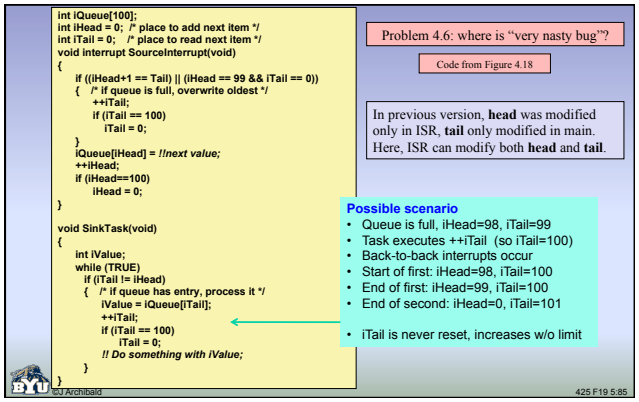
Problem 4.5: The task and interrupt code share the fTaskCodeUsingTempsB variable.

Is the task's use of this variable (fTaskCodeUsingTempsB) atomic? Does it need to be atomic for the code to work correctly?


```
int iQueue[100];
int iHead = 0;  /* place to add next item */
int iTail = 0;  /* place to read next item */
void interrupt SourceInterrupt(void)
{
    if ((iHead+1 == Tail) || (iHead == 99 && iTail == 0))
    {   /* if queue is full, overwrite oldest */
        ++iTail;
        if (iTail == 100)
            iTail = 0;
    }
    iQueue[iHead] = !!next value;
    ++iHead;
    if (iHead==100)
        iHead = 0;
}

void SinkTask(void)
{
    int iValue;
    while (TRUE)
        if (iTail != iHead)
        {   /* if queue has entry, process it */
            iValue = iQueue[iTail];
            ++iTail;
            if (iTail == 100)
                iTail = 0;
            !! Do something with iValue;
        }
}
```

Problem 4.6: where is "very nasty bug"?

Code from Figure 4.18


int iQueue[100];
int iHead = 0; /* place to add next item */
int iTail = 0; /* place to read next item */
void interrupt SourceInterrupt(void)
{
if ((iHead+1 == Tail) || (iHead == 99 && iTail == 0))
{ /* if queue is full, overwrite oldest */
++iTail;
if (iTail == 100)
iTail = 0;
}
iQueue[iHead] = !!next value;
++iHead;
if (iHead==100)
iHead = 0;
}
void SinkTask(void)
{
int iValue;
while (TRUE)
if (iTail != iHead)
{ /* if queue has entry, process it */
iValue = iQueue[iTail];
++iTail;
if (iTail == 100)
iTail = 0;
!! Do something with iValue;
}
}
Problem 4.6: where is "very nasty bug"?
Code from Figure 4.18
In previous version, head was modified only in ISR, tail only modified in main. Here, ISR can modify both head and tail.
Possible scenario
• Queue is full, iHead=98, iTail=99
• Task executes ++iTail (so iTail=100)
• Back-to-back interrupts occur
• Start of first: iHead=98, iTail=100
• End of first: iHead=99, iTail=100
• End of second: iHead=0, iTail=101
• iTail is never reset, increases w/o limit
©J Archibald
425 F19 5:85