
Process Coordination and Shared Data

Lecture 26



In These Notes . . .

Sharing data safely

- When multiple threads/processes interact in a system, new species of bugs arise
 1. Compiler tries to save time by not reloading values which it doesn't realize may have changed
 2. Switching between threads can lead to trying to operate upon partially updated variables/data structures
- We must design the system to prevent or avoid them

Operating System support for Process Coordination

- Monitors
- When multiple thread/processes interact in a system, new species of bugs arise
- We must design the system to prevent or avoid them
- Bugs and solutions

Volatile Data

Compilers assume that variables in memory do not change spontaneously, and optimize based on that belief

- *Don't reload a variable from memory if you haven't stored a value there*
- Read variable from memory into register (faster access)
- Write back to memory at end of the procedure, or before a procedure call

This optimization can fail

- Example: reading from input port, polling for key press
 - `while (SW_0) ;` will read from `SW_0` once and reuse that value
 - Will generate an infinite loop triggered by `SW_0` being true

Variables for which it fails

- Memory-mapped peripheral register – register changes on its own
- Global variables modified by an ISR – ISR changes the variable
- Global variables in a multithreaded application – another thread or ISR changes the variable

The Volatile Directive

Need to tell compiler which variables may change outside of their control

- Use volatile keyword to force compiler to reload these vars from memory for each use

```
volatile unsigned int num_ints;
```

- Pointer to a volatile int

```
volatile int * var; // or  
int volatile * var;
```

- Now each C source read of a variable (e.g. status register) will result in a assembly language move instruction
- Good explanation in Nigel Jones' "Volatile," *Embedded Systems Programming* July 2001

Cooperation and Sharing Information

Program consists of one or more threads/processes

Any two threads/processes are either independent or cooperating

Cooperation enables

- Improved performance by overlapping activities or working in parallel
- Better program structure (easier to develop and debug)
- Easy sharing of information

Two methods to share information

- Shared memory
- Message passing

Shared Memory

Is practical when communication cost is low

Low-end embedded systems have no memory protection support

- Threads can access the data directly – e.g. global variables
- (Who needs seatbelts or airbags!)

UNIX and high-end embedded systems have memory protection support

- Impossible to see other processes' memory space by default
 - E.g. virtual memory
- Establish a mapping between process's address space to a named memory object which can be shared across processes
- POSIX Threads (pthreads) API is a standard for workstation programming

Message Passing

Most useful when communication cost is high

- Often used for distributed systems

Producer process generates message, *consumer* process receives it

Each process must be able to name other process

Consumer is assumed to have an infinite receive queue

- Bounded queue complicates the programming

OS manages messages

Mailbox is a queue with only one entry

The Shared Data Problem

Often we want to split work between
ISR and the task code

Some variables must be shared to
transfer information

Problem results from task code using
shared data *non-atomically*

- An *atomic* part of a program is non-interruptible
- A *critical section* (group of instructions) in a program **must** be executed atomically for correct program behavior

get_ticks() returns a long, formed by
concatenating variable tchi and
register tc

- If an interrupt occurs in get_ticks,
we may get **old** value of tchi and
new value of tc



```
volatile unsigned int
tchi=0, tc=0;
#pragma INTERRUPT tc_isr
void tc_isr(void) {
    tc++; if(!tc) tchi++;
}
```

```
unsigned long get_ticks(){
    unsigned long temp;
    temp = tchi;
    temp <<= 16;
    temp += tc;
    return temp;
}
```

Step	temp	tchi	tc
1	0x00000123	0x0123	0xffff
2	0x01230000	0x0123	0xffff
3	0x01230000	0x0124	0x0000
4	0x0123000	0x0124	0x0000

Critical Sections Lead to Race Conditions

Critical section: A **non-re-entrant** piece of code that can only be executed by one process at a time. Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use.

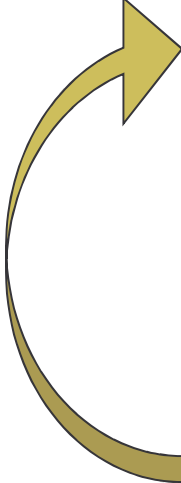
Re-entrant Code: Code which can have multiple simultaneous, interleaved, or nested invocations which will not interfere with each other. This is important for parallel processing, recursive functions or subroutines, and interrupt handling.

- If invocations must share data, the code is non-reentrant. (e.g. using global variable, not restoring all relevant processor state (e.g. flags))
- If each invocation has its own data, the code is reentrant. (e.g. using own stack frame and restoring all relevant processor state)

Race condition: Anomalous behavior due to unexpected critical dependence on the relative timing of events. Result of increment example depends on the *relative timing* of the read and write operations.

Long Integer

```
long int ct;
void f1() {
    ct++;
}
void f2() {
    if (ct==0x10000)
        /* ... */
}
```



```
; void f1()
add.w  #0001H,_ct
adcf.w _ct+2
rts

; void f2()
cmp.w  #0,_ct
jnz    unequal
cmp.w  #1,_ct+2
jnz    unequal
; equal
unequal:
; unequal
```

What if f2() starts running after the f1's `add.w` (resulting in a carry) but before the `adcf.w`?

Race condition due to **non-atomic** operation

- Data structures
- Large variables

Is Queue Access Atomic for Serial Example?

Size field is modified by both enqueue and dequeue functions

Does compiler generate code which is atomic?

This code is very inefficient – the compiler vendor wants you to buy the licensed and optimized version

```
; Enqueue
; q->Size++;
    mov.w -2[FB],A0      ; q
    mov.w -2[FB],A1      ; q
    mov.w 0024H[A0],0024H[A1]
    add.w #0001H,0024H[A1]
```

```
; Dequeue
; q->Size--;
    mov.w -3[FB],A0      ; q
    mov.w -3[FB],A1      ; q
    mov.w 0024H[A0],0024H[A1]
    sub.w #0001H,0024H[A1]
```

Solution 1 – Disable Interrupts

Disable interrupts during critical section

- Renesas syntax ->

Problems

- **You** must determine where the critical sections are, not the compiler (it's not smart enough)
- Disabling interrupts increases the response time for other interrupts
- What if interrupts were already disabled when we called `get_ticks`?
- Need to restore the interrupt masking to previous value

```
#define ENABLE_INTS
    {_asm(" FSET I");}
#define DISABLE_INTS
    {_asm(" FCLR I");}

unsigned long get_ticks(){
    unsigned long temp;
    DISABLE_INTS;
    temp = tchi;
    temp <<= 16;
    temp += tc;
        ENABLE_INTS;
    return temp;
}
```



Are Interrupts Currently Enabled?

FLG's I flag (bit 6)

- Enables/disables interrupts
- Section 1.4 of ESM

Need to examine flag register, but how?

- Not memory-mapped
- Can't access with BTST

Solution

- STC: Store from control register (ESM, p. 123)
- Use a macro (CLPM, p. 98) to copy the flag bit into a variable **iflg** in our code (we copy the whole register, then mask out the other bits) – *nifty feature!*
- Later use that variable **iflg** to determine whether to re-enable interrupts

```
#define I_MASK (0x0040)
#define GET_INT_STATUS(x) {_asm(" STC
FLG,$$[FB]",x); x &= I_MASK;}
#define ENABLE_INTS {_asm(" FSET I");}
#define DISABLE_INTS {_asm(" FCLR I");}
```

```
unsigned long get_ticks(){
    unsigned long temp, iflg;
    GET_INT_STATUS(iflg);
    DISABLE_INTS;
    temp = tchi;
    temp <<= 16;
    temp += tc;
    if (iflg)
        ENABLE_INTS;
    return temp;
}
```

Solution 2 – Repeatedly Read Data

Keep reading until the function returns the same value

- Easy here because `get_seconds` returns an easily compared value (a long)

Problems which limit this approach

- `tc` might be changing every clock cycle, so `get_ticks` would never return. *Loop time must be short compared with interrupt frequency*
- What if we wanted to compare two structures? Would need a function (slower, more code)
- Compiler may optimize out code

```
unsigned long get_seconds() {
    unsigned long temp1, temp2;

    temp2 = tchi;
    temp2 <<= 16;
    temp2 += tc;
    do {
        temp1 = temp2;
        temp2 = tchi;
        temp2 <<= 16;
        temp2 += tc;
    } while (temp1 != temp2);
    return temp2;
}
```



A Gotcha! TC keeps changing!

See Ganssle's "Asynchronicity"

Solution: after disabling interrupts, do the timer C ISR's work if needed

Examine Interrupt Request bit of tcic (timer C interrupt control register), which indicates overflow

Increment counter if it did overflow

```
unsigned long get_ticks(){
    unsigned long temp, iflg;
    unsigned temp1, temp2;
    GET_INT_STATUS(iflg);
    DISABLE_INTS;
    temp2 = tc;
    temp1 = tchi;
    if (ir_tcic) {
        temp1++;
        temp2 = tc;
    }
    if (iflg)
        ENABLE_INTS;
    temp = temp1;
    temp <<= 16;
    temp += temp2;
    return temp;
}
```

Solution 3 – Use a Lock

Relies on kernel/scheduler for efficiency

Define a lock variable (global) for each resource to be shared (variable (inc. data structure), I/O device)

- Lock is 0 if resource is available
- Lock is 1 if resource is busy

Functions agree to check lock before accessing resource

- if lock is 0, can use resource
- if lock is 1, need to try again later
 - if preemptive kernel is used, call kernel to reschedule this thread later
 - for non-preemptive kernel, call kernel to yield processor to other threads

Enable interrupts when possible to reduce interrupt latency

Some processors have atomic read-modify-write instructions, avoiding need to disable interrupts when accessing lock variable

```
DISABLE_INTS
if (lock_var == 0) {
    lock_var = 1;
    ENABLE_INTS
    access resource
    DISABLE_INTS
    lock_var = 0;
    ENABLE_INTS
} else {
    ENABLE_INTS
    // try again
    later
}
```


Atomic Read-Modify-Write Instructions

Test-and-set

- Read a memory location and, if the value is 0, set it to 1 and return true. Otherwise, return false
- M16C: *BTSTS dest* (Bit test and set)
 - $Z \leq 1$ if $dest == 0$ (“return value is Z flag”), else $Z \leq 0$
 - $C \leq 1$ if $dest \neq 0$, else $C \leq 0$
 - $dest \leq 1$
- BTSTC: Bit test and clear

Fetch-and-increment

- Return the current value of a memory location and increment the value in memory by 1

Compare-and-swap

- Compare the value of a memory location with an old value, and if the same, replace with a new value

Load-Locked, Store-Conditional

Load-Linked, Store-Conditional (LLSC)

- Pair of instructions may be easier to implement in hardware
- Load-linked (or load-locked) returns the value of a memory location
- Store-conditional stores a new value to the same memory location if the value of that location has not been changed since the LL. Returns 0 or 1 to indicate success or failure
- If a thread is switched out between an LL and an SC, then the SC automatically fails

Simple Spin Lock

Test-and-set

```
spin_lock(lock) {  
    while (test-and-set(lock) == FALSE);  
}
```

```
spin_unlock(lock){  
    lock = 0;  
}
```

Simple, but slow and wastes time

- Requires OS to switch out this thread eventually and resume another, which will eventually let spin_lock finish (we hope)

Typically use an OS call to improve efficiency, as OS knows immediately if lock is available

- If available, grant lock to requesting thread and resume execution
- If not available, move requesting thread to wait queue and resume next thread

Solution 4 – Disable the Scheduler

If no ISR shares this data with the thread, can disable scheduler, keeping it from switching to another thread

Interrupts are still enabled

Counter-productive

- We added the scheduler to provide efficient processor sharing
- This defeats the purpose of the scheduler!

Solution 5 – Use an OS Semaphore

Operating system typically offers mutual exclusion support through **semaphores**

- Provide mutually exclusive access to a shared resource
- Signal occurrence of events
- Link resumption of threads to semaphore events
- Allow tasks to synchronize their activities

Behavior

- Thread requests semaphore to enter critical section
- If semaphore available (non-zero), thread enters critical section and OS updates semaphore state (sets to zero or decrements)
- If semaphore unavailable (zero), OS moves thread to waiting queue
- When a semaphore becomes available, OS moves the thread waiting on it to the ready queue
- After critical section, thread releases semaphore

Semaphore Operations by OS

Creation/initialization

Take/Wait/Pend/P

- Often includes time-out parameter. Wait returns error code, allowing calling task to decide how to deal with lack of semaphore.

Release/Signal/Post/V

- If no task is waiting on semaphore, increment its value
- If any tasks are waiting on this semaphore, move the highest priority (or longest-waiting) task to the Ready queue

Two types of Semaphores

- Binary (0 and 1)
 - Only one thread can access shared resource at a time
- Counting (0 through N)
 - Up to N devices can access shared resource at a time

Using Semaphores

Rules and Overview

- We create a semaphore to guard a shared resource to maintain data integrity
- We must get permission to access the resource
- We must release that permission when done

Semaphore operations

- Take (P) the semaphore before (down, pend)
- Release (V) it after (up, post)

Value of semaphore indicates number of units of resource available for use

- Use a binary semaphore (1 or 0) to control access to a specific resource

P: wait until semaphore is free, then *take* it (down)

- If semaphore is free, take it and continue executing
- Otherwise put calling thread into *waiting* state

V: *release* the semaphore (up)

- If a task is waiting for this semaphore, move that task to the ready queue

```
long int counter;
void f1() {
    Take(counter_sem);
    counter++;
    Release(counter_sem);
}
void f2() {
    Take(counter_sem);
    counter++;
    Release(counter_sem);
}
```

Solutions to Shared Data Problem

1. Disable task switches
 - No effect on response time for interrupts
 - Doesn't handle ISRs
2. Disable interrupts
 - Only method if ISR and task share data
 - Fast – single instruction, typically
 - Greedy – slows down response time for all other threads
3. Use a lock variable
 - Poor performance if no kernel used
4. Disable scheduler
 - Poor performance if no kernel used
5. Use OS-provided semaphore
 - Some slowdown, but only significantly affects threads using them
 - Need more software

Monitors

Semaphores have a few limitations: unstructured, difficult to program correctly. Monitors eliminate these limitations and are as powerful as semaphores

A monitor consists of a software module with one or more procedures, an initialization sequence, and local data (can only be accessed by procedures)

Structure

- The critical section of each concurrent task is replaced by a call to the **monitor** operation
- An implicit semaphore is associated with each **monitor**, called the **monitor lock**

Rules

- User doesn't directly access monitor lock
- Only one task is active in the **monitor** at any one time
- A call to a **monitor** operation results in the calling task acquiring the associated semaphore
- If the lock is already taken, the calling task blocks until the lock is acquired
- An exit from the **monitor** operation releases the semaphore -- the **monitor lock** is released so it can be acquired by a different task

Monitors and Programming Languages

Where are they?

- Most programming languages do not specify concurrency and synchronization mechanisms, must be added
- Some do: Java, Concurrent Pascal, Modula 2, Modula 3

Details

- Identify method as a critical section using **synchronized** keyword
- The Java compiler inserts code to
 - Get lock immediately after entering increment()
 - Release lock immediately before returning from it

```
class Counter {  
    long value=0;  
    public synchronized void  
        increment() {  
        value++;  
    }  
}
```

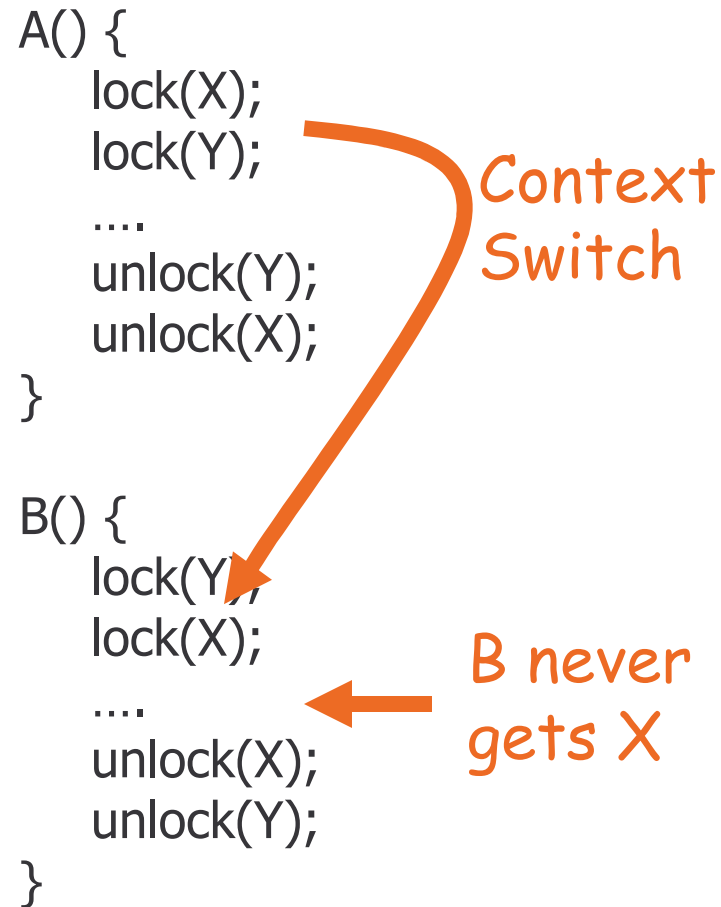
Deadlock

A needs resources X and Y

B needs resources X and Y

Sequence leading to deadlock

- A requests and gets (locks) X
- context switch
- B locks Y
- B requests X, doesn't get it, leading to...
- context switch
- A can't get Y
- B can't get X



Deadlock (Cont'd)

Deadlock: A situation where two or more processes are unable to proceed because each is waiting for one of the others to do something.

Livelock: When two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work. This is similar to deadlock in that no progress is made but differs in that neither process is blocked or waiting for anything.

Deadlock can occur whenever multiple parties are competing for exclusive access to multiple resources -- what can be done?

- Deadlock prevention
- Deadlock avoidance
- Deadlock detection and recovery

Deadlock Prevention

Deny one of the four necessary conditions

- Make resources sharable
 - *No mutual exclusion*
- Processes MUST request ALL resources at the same time.
 - Either all at start or release all before requesting more
 - “Hold and wait for” not allowed
 - *Poor resource utilization and possible starvation*
- If process requests a resource which is unavailable
 - It must release all resources it currently holds and try again later
 - Allow preemption
 - *Leads loss of work*
- Impose an ordering on resource types.
 - Process requests resources in a pre-defined order
 - No circular wait
 - *This can be too restrictive*

More Deadlock Strategies

Avoidance

- Allow necessary conditions to occur, but use algorithms to predict deadlock and refuse resource requests which could lead to deadlock – Called Banker's Algorithm
- *Running this algorithm on all resource requests eats up compute time*

Detection and Recovery

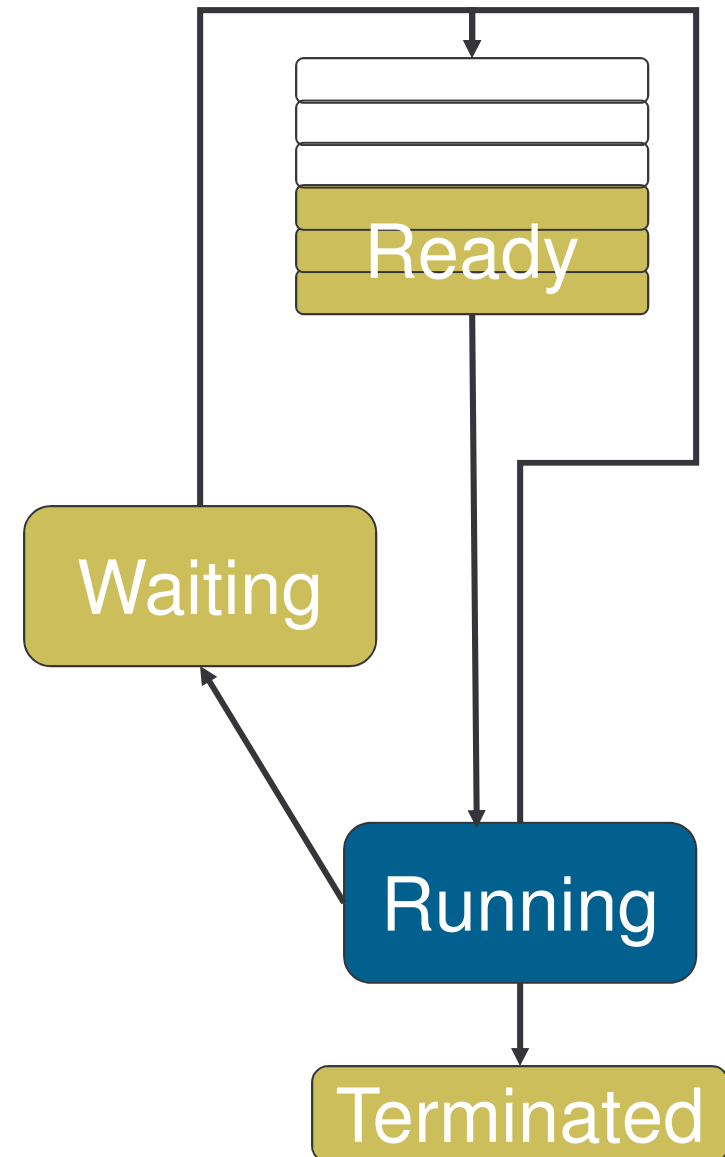
- Check for circular wait periodically. If detected, terminate all deadlocked processes (extreme solution but very common)
- *Checking for circular wait is expensive*
- *Terminating all deadlocked processes might not be appropriate*

Scheduling

Choosing which *ready* thread to run next

Common criteria

- CPU Utilization – fraction of time is the CPU busy
- Throughput – number of tasks are completed per unit time
- Turnaround time – time delay from task first being submitted to OS to finally completing
- Waiting time – amount of time a task spends in waiting queue
- Response time – time delay from request submission to first processing in response to that request



Common Scheduling Algorithms

First-Come, First Served (FCFS)

- All queues operate as strict FIFOs without priority
- Problems: large average delay, not preemptive

Round Robin: add time-sharing to FCFS

- At end of time tick, move currently running task to end of ready queue
- Problems: Still have a large average delay, choosing time-tick is trade-off of context-switching overhead vs. responsiveness

Shortest Job First (SJF)

- Job = process
- SJF is provably optimal in minimizing average waiting time
- Problem: How do we determine how long the next job will take?
 - Could predict it based on previous job?

Priority Scheduling

Run the ready task with highest priority

Define priority

- Internal: Time limits, memory requirements
- External: Importance to application, fees paid, department submitting task

Problem: indefinite blocking (starvation)

- Low level processes may never get to run in heavily loaded system
- Two outcomes
 - Processes run during winter break
 - Processes disappear when computer eventually crashes



From OS to RTOS

Traditional (non-real-time) Operating System

- Hard to predict response time...
- Hard to guarantee that a task will always run before its deadline

Real-Time Operating System

- Easy to determine that a task will always run before its **deadline**
- Designed for **periodic** tasks

What does Real-Time mean?



Late
answers
are wrong
answers!



Scheduling – Selecting a *Ready* task to run

Goals

- Meet all task deadlines
- Maximize processor *utilization* (U)
 - U = Fraction of time CPU performs useful work
 - Limit scheduling overhead (choosing what to run next)
 - Limit context switching overhead

Assigning priority based *only* on importance doesn't work – why not?

How do we assign priorities to task?

- Statically – priority based on period (doesn't change)
- Dynamically – priority based on time left (changes)

Definitions for Task i

- Task execution time = T_i
- Task execution period = τ_i : time between arrivals
- Utilization = fraction of time which CPU is used

- For a task i

$$U_i = \frac{T_i}{\tau_i}$$

- Overall, for all n tasks in the system

$$U = \sum_{i=1}^n \frac{T_i}{\tau_i}$$

- Completion Time = time at which task finishes
- Critical Instant = time at which task's completion time is maximized. All tasks arrive simultaneously.
- Schedulable = a schedule exists which allows all tasks to meet their deadlines, even for the critical instant

Rate Monotonic Scheduling

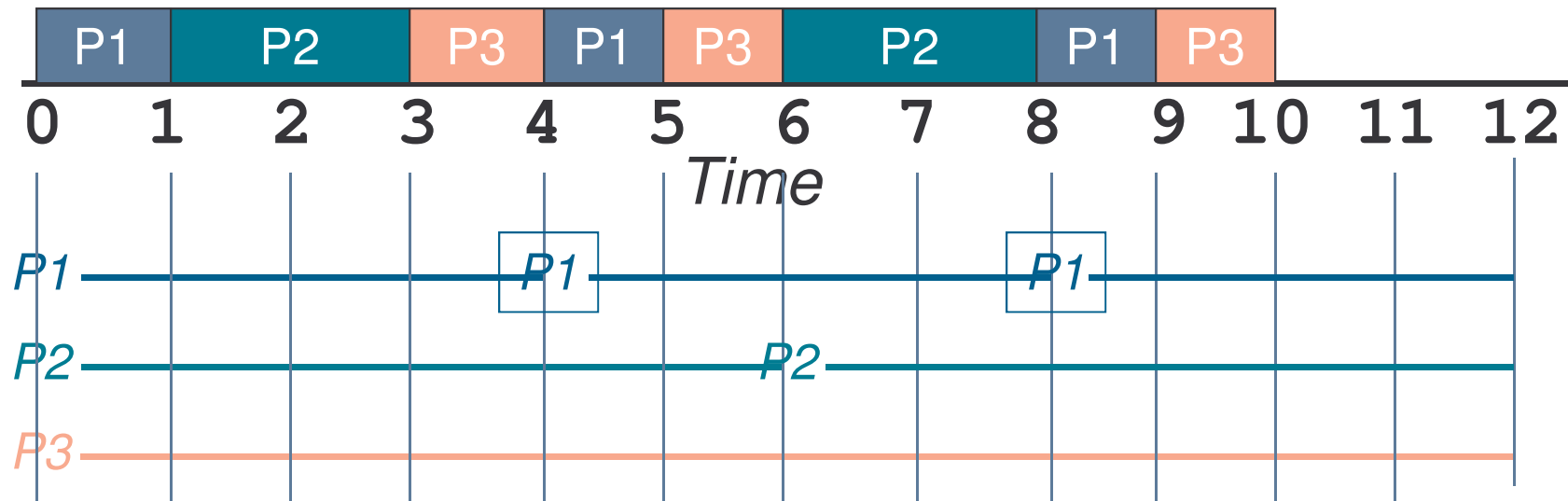
Assumptions

- Tasks are periodic with period τ_i
- Single CPU
- $T_{ContextSwitch} = T_{scheduler} = 0$
- No data dependencies between tasks
- Constant process execution time T_i
- Deadline = end of period = τ_i

Assign priority based on period (rate)

- Shorter period means higher priority

Processor Behavior – Graphical Analysis



Task	Exec. Time T	Period t	Priority
P1	1	4	High
P2	2	6	Medium
P3	3	12	Low

Exact Schedulability Test for Task i

Account for all processing at critical instant

Consider possible additional task arrivals

$a_n = n$ th estimate of time when task i completes

Loop

- Estimate higher priority job arrivals, compute completion time
- Recompute based on any new arrivals

Iterate until

- $a_n > \tau_i$: not schedulable
- $a_n = a_{n-1} \leq \tau_i$: schedulable

$$a_0 = \sum_{j=0}^i T_j$$

$$a_{n+1} = T_i + \sum_{j=0}^{i-1} \left\lceil \frac{a_n}{\tau_j} \right\rceil T_j$$

Exact Schedulability Test for Example

$$a_0 = \sum_{j=0}^i T_j = 1 + 2 + 3 = 6$$

$$a_1 = 3 + \sum_{j=0}^{i-1} \left\lceil \frac{6}{\tau_j} \right\rceil T_j = 3 + \left\lceil \frac{6}{4} \right\rceil * 1 + \left\lceil \frac{6}{6} \right\rceil * 2 = 3 + 2 + 2 = 7$$

$$a_2 = 3 + \sum_{j=0}^{i-1} \left\lceil \frac{7}{\tau_j} \right\rceil T_j = 3 + \left\lceil \frac{7}{4} \right\rceil * 1 + \left\lceil \frac{7}{6} \right\rceil * 2 = 3 + 2 + 4 = 9$$

$$a_3 = 3 + \sum_{j=0}^{i-1} \left\lceil \frac{9}{\tau_j} \right\rceil T_j = 3 + \left\lceil \frac{9}{4} \right\rceil * 1 + \left\lceil \frac{9}{6} \right\rceil * 2 = 3 + 3 + 4 = 10$$

$$a_4 = 3 + \sum_{j=0}^{i-1} \left\lceil \frac{10}{\tau_j} \right\rceil T_j = 3 + \left\lceil \frac{10}{4} \right\rceil * 1 + \left\lceil \frac{10}{6} \right\rceil * 2 = 3 + 3 + 4 = 10$$

Iterate until $a_{n-1} = a_n$ $a_3 = a_4 < 12$, so system is schedulable

Utilization Bound for RMS

Utilization U for n tasks

- Fraction of time spent on tasks

Maximum utilization U_{Max} for m tasks

- Max. value of U for which we can guarantee RMS works

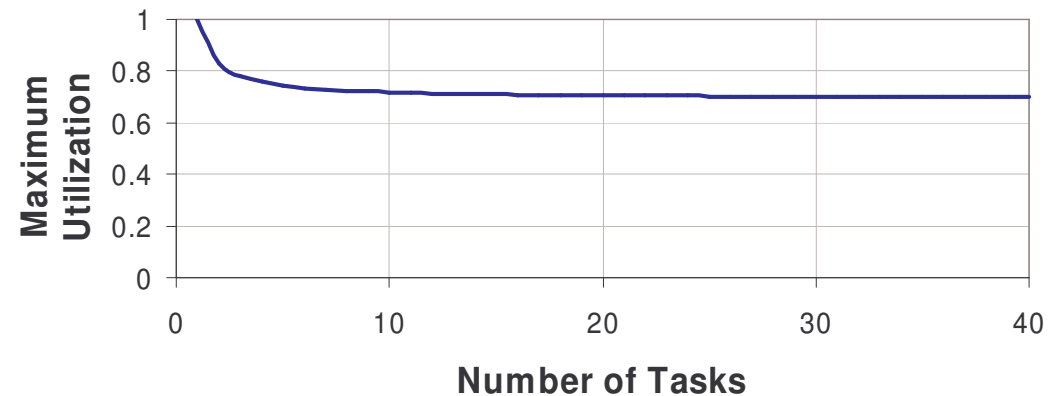
$$U = \sum_{i=1}^n \frac{T_i}{\tau_i}$$

Utilization bound test

- $U < U_{Max}$: always schedulable with RMS
- $U_{Max} < U < 1.0$: inconclusive
- $U > 1.0$: Not schedulable

$$U_{Max} = m \left(2^{1/m} - 1 \right)$$

Why is U_{Max} so small?
(approaches $\ln(2)$)
Conservative



Example of Scheduling with RMS and UB

Task	Exec. Time T	Period τ	Priority
P1	1	4	High
P2	2	6	Medium
P3	3	12	Low

$$U = \frac{T_1}{\tau_1} + \frac{T_2}{\tau_2} + \frac{T_3}{\tau_3} = \frac{1}{4} + \frac{2}{6} + \frac{3}{12} = 0.833$$

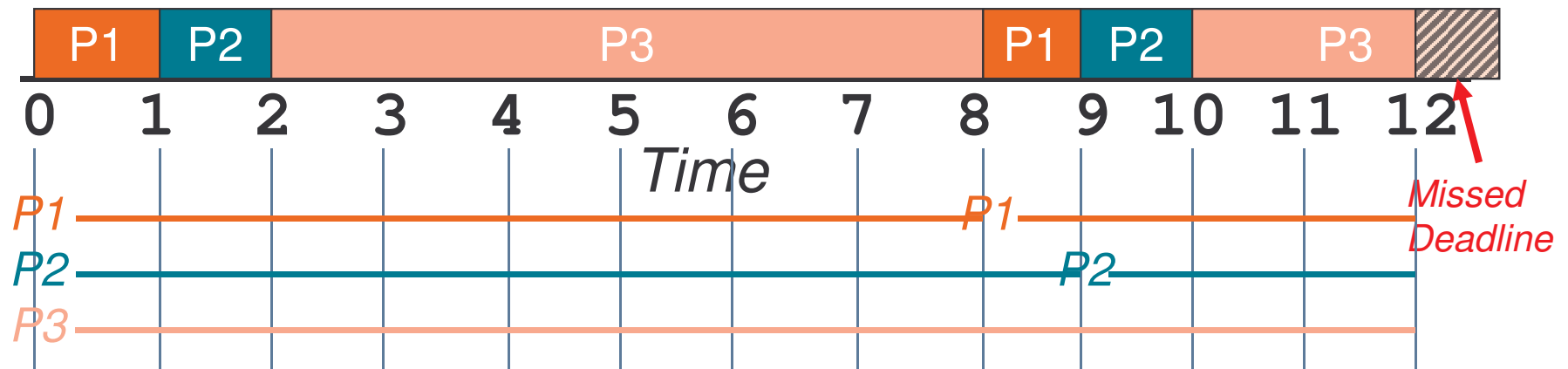
$$U_{Max} = m(2^{1/m} - 1) = 3(2^{1/3} - 1) = 0.780$$

Utilization Bound test is inconclusive

RMS Sometimes Fails Under 100% Utilization

For some workloads with utilization below 100%, RMS priority allocation can fail
Tasks P1, P2 have later deadlines than P3 yet preempt it due to their shorter periods

Thread	Exec. Time T	Period τ	Priority
P1	1	8	High
P2	1	9	Medium
P3	9	12	Low



Counter-example provided by C. Palenchar

Earliest Deadline First

Can guarantee schedulability at up to 100% utilization

Can't use Exact Schedulability Test for EDF

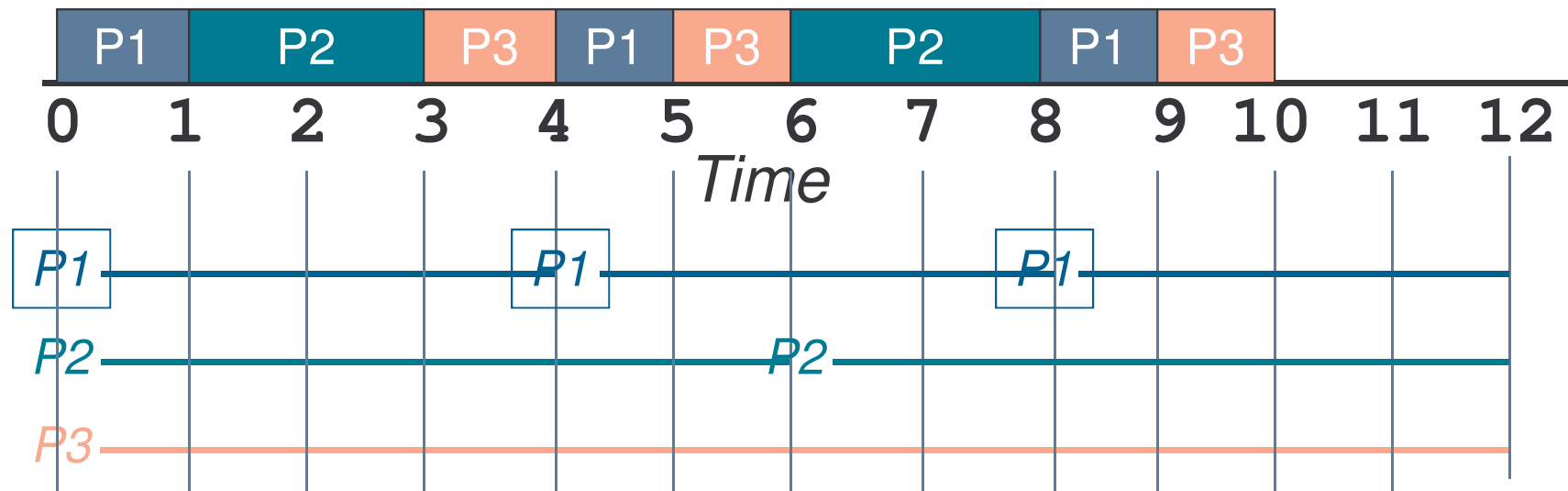
- Sum up all possible higher priority tasks, but priority depends on how close deadlines are!
- Can we modify the test to deal with this?

How does the kernel keep track of upcoming deadlines?

- Can determine priority when inserting task into ready queue
 - Need to search through queue to find correct location (based on deadline)
- Can determine which task to select from ready queue
 - Need to search through queue to find earliest deadline
- Both are up to $O(n)$ search time
 - Can also do binary search tree

Earliest Deadline First Example

Thread	Execution Time T	Period τ
P1	1	4
P2	2	6
P3	3	12



System Performance During Transient Overload

RMS – Each task has fixed priority. *So?*

- This priority determines that tasks will be scheduled consistently
 - Task A will always preempt task B if needed
 - Task B will be forced to miss its deadline to help task A meet its deadline

EDF – Each task has varying priority. *So?*

- This priority depends upon when the task's deadline is, and hence when the task becomes ready to run (*arrival time*)
 - Task B may have higher priority than A depending on arrival times
 - To determine whether task A or B will miss its deadline we need to know their arrival times