Concurrency (I)
--- Windows Traps

Roadmap for This Lecture

- Concurrency Review
  - Critical-section Problem
  - Software solution
  - Hardware solution
  - Semaphores
  - Deadlocks and Starvation
- Windows Traps
  - Interrupts
  - Deferred Procedure Calls (DPCs)
  - Asynchronous Procedure Calls (APCs)
  - Exception Dispatching
  - System Service Dispatching

The Critical-Section Problem

- *n* threads all competing to use a shared resource (i.e.; shared data)
- Each thread has a code segment, called *critical section*, in which the shared data is accessed
- Problem:
  Ensure that when one thread is executing in its critical section, no other thread is allowed to execute in its critical section

Three Requirements

1. Mutual Exclusion
   - Only one thread at a time is allowed into its critical section, among all threads that have critical sections for the same resource or shared data.
2. Progress
   - If no thread is in the critical section and some threads want to enter, then only those threads not in the remainder section can participate in the decision of which thread gets to enter next.
   - The selection process cannot be postponed indefinitely.
3. Bounded Waiting
   - There must be a bound on the number of times that other processes are allowed to enter their critical section after a process has requested to enter its critical section and before the request is granted.

Initial Attempts to Solve Problem

- Only 2 threads, T_0 and T_1
- General structure of thread T_i (other thread T_j)

```
do {
    enter section
        critical section
    exit section
        reminder section
} while (1);
```

- Threads may share some common variables to synchronize their actions.

First Attempt: Algorithm 1

- Shared variables - initialization

```
int turn = 0;
```

- `turn == i` ⇒ T_i can enter its critical section
- Thread T_i

```
do {
    while (turn != i) ;
        critical section
    turn = j;
        reminder section
} while (1);
```

- Strict alternation between i and j
- Satisfies mutual exclusion, but not progress

Second Attempt: Algorithm 2

- Shared variables - initialization

```
int flag[2]; flag[0] = flag[1] = 0;
```

- `flag[i] == 1` $\Rightarrow$ T_i can enter its critical section
- Thread T_i

```
do {
        flag[i] = 1;
        while (flag[j] == 1) ;
                    critical section
        flag[i] = 0;
                remainder section
} while(1);
```

- Satisfies mutual exclusion, but not progress requirement.
- Very sensitive to timing of the two threads

Third Attempt: Algorithm 3
(Peterson's Algorithm - 1981)

- Shared variables of algorithms 1 and 2 - initialization:

```
int flag[2]; flag[0] = flag[1] = 0;
int turn = 0;
```

- Thread T_i

```
do {

        flag[i] = 1;
        turn = j;
        while ((flag[j] == 1) && turn == j) ;

              critical section

        flag[i] = 0;

              remainder section

} while (1);
```

- Solves the critical-section problem for two threads.

Dekker's Algorithm (1965)

- This is the first correct solution proposed for the two-thread (two-process) case.
- Originally developed by Dekker in a different context, it was applied to the critical section problem by Dijkstra.
  - Dekker adds the idea of a favored thread and allows access to either thread when the request is uncontested.
  - When there is a conflict, one thread is favored, and the priority reverses after successful execution of the critical section.

Dekker's Algorithm (contd.)

- Shared variables - initialization:

```
int flag[2]; flag[0] = flag[1] = 0;
int turn = 0;
```

- Thread T_i

```
do {
       flag[i] = 1;
       while (flag[j] )
              if (turn == j) {
                     flag[i] = 0;
                     while (turn == j);
                     flag[i] = 1;
              }
                     critical section
       turn = j;
       flag[i] = 0;
                     remainder section
} while (1);
```

Bakery Algorithm
(Lamport 1979)

A Solution to the Critical Section problem for n threads

- Before entering its critical section, a thread receives a number. Holder of the smallest number enters the critical section.
- If threads T_i and T_j receive the same number,
  if $i < j$, then T_i is served first; else T_j is served first.
- The numbering scheme generates numbers in monotonically non-decreasing order;
  i.e., 1,1,1,2,3,3,3,4,4,5...

Bakery Algorithm

- Notation “<“ establishes lexicographical order among 2-tuples (ticket #, thread id #)

  $(a,b) < (c,d)$ if $a < c$ or if $a == c$ and $b < d$

  $\max(a_0,\ldots, a_{n-1}) = \{ k \mid k \geq a_i \text{ for } i = 0,\ldots, n-1 \}$

- Shared data

```
int choosing[n];
int number[n];
```
  \- the ticket (for `number[n]`)

Data structures are initialized to 0

Bakery Algorithm

```
do {
   choosing[i] = 1;
  number[i] = max(number[0],number[1] ...,number[n-1]) + 1;
   choosing[i] = 0;
   for (j = 0; j < n; j++) {
       while (choosing[j] == 1) ;
       while ((number[j] != 0) &&
              ((number[j],j) ''<'' (number[i],i)));
   }
       critical section
   number[i] = 0;
       remainder section
} while (1);
```

Mutual Exclusion - Hardware Support

- Interrupt Disabling
  - Concurrent threads cannot overlap on a uniprocessor
  - Thread will run until performing a system call or interrupt happens
- Special Atomic Machine Instructions
  - Test and Set Instruction - read & write a memory location
  - Exchange Instruction - swap register and memory location
- Problems with Machine-Instruction Approach
  - Busy waiting
  - Starvation is possible
  - Deadlock is possible

Synchronization Hardware

- Test and modify the content of a word atomically

```
boolean TestAndSet(boolean &target) {
  boolean rv = target;
  target = true;

  return rv;
}
```

Mutual Exclusion with Test-and-Set

- Shared data:

```
boolean lock = false;
```

- Thread T_i

```
do {
  while (TestAndSet(lock)) ;
    critical section
  lock = false;
    remainder section
}
```

Synchronization Hardware

- Atomically swap two variables.

```
void Swap(boolean &a, boolean &b) {
  boolean temp = a;
  a = b;
  b = temp;
}
```

Mutual Exclusion with Swap

- Shared data (initialized to **0**):

```
int lock = 0;
```

- Thread T_i

```
int key;
do {
    key = 1;
    while (key == 1) Swap(lock, key);
        critical section
    lock = 0;
        remainder section
}
```

Semaphores

- Semaphore *S* – integer variable
- can only be accessed via two atomic operations

```
wait (S):
        while (S <= 0);
        S--;

signal (S):
        S++;
```

Critical Section of n Threads

- Shared data:

```
semaphore mutex; //initially mutex = 1
```

- Thread T_i:

```
do {
      wait(mutex);
            critical section
      signal(mutex);
            remainder section
} while (1);
```

Semaphore Implementation

- Semaphores may suspend/resume threads
  - Avoid busy waiting
- Define a semaphore as a record

```
typedef struct {
    int value;
    struct thread *L;
} semaphore;
```

- Assume two simple operations:
  - **suspend()** suspends the thread that invokes it.
  - **resume(*T*)** resumes the execution of a blocked thread **T**.

Implementation

- Semaphore operations now defined as

wait(*S*):

```
S.value--;
if (S.value < 0) {
        add this thread to S.L;
        suspend();
}
```

signal(*S*):

```
S.value++;
if (S.value <= 0) {
        remove a thread T from S.L;
        resume(T);
}
```

Semaphore as a General Synchronization Tool

- Execute *B* in T_j only after *A* executed in T_i
- Use semaphore *flag* initialized to 0
- Code:

T_i	T_j
…	…
A	*wait*(*flag*)
signal(*flag*)	*B*

Two Types of Semaphores

- *Counting* semaphore
  - integer value can range over an unrestricted domain.
- *Binary* semaphore
  - integer value can range only between 0 and 1;
  - can be simpler to implement.
- Counting semaphore *S* can be implemented as a binary semaphore.

Deadlock and Starvation

- **Deadlock** – two or more threads are waiting indefinitely for an event that can be caused by only one of the waiting threads.
- Let *S* and *Q* be two semaphores initialized to 1

T_0	T_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
. . .	. . .
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- **Starvation** – indefinite blocking. A thread may never be removed from the semaphore queue in which it is suspended.
- Solution - all code should acquire/release semaphores in same order

Getting Into Kernel Mode

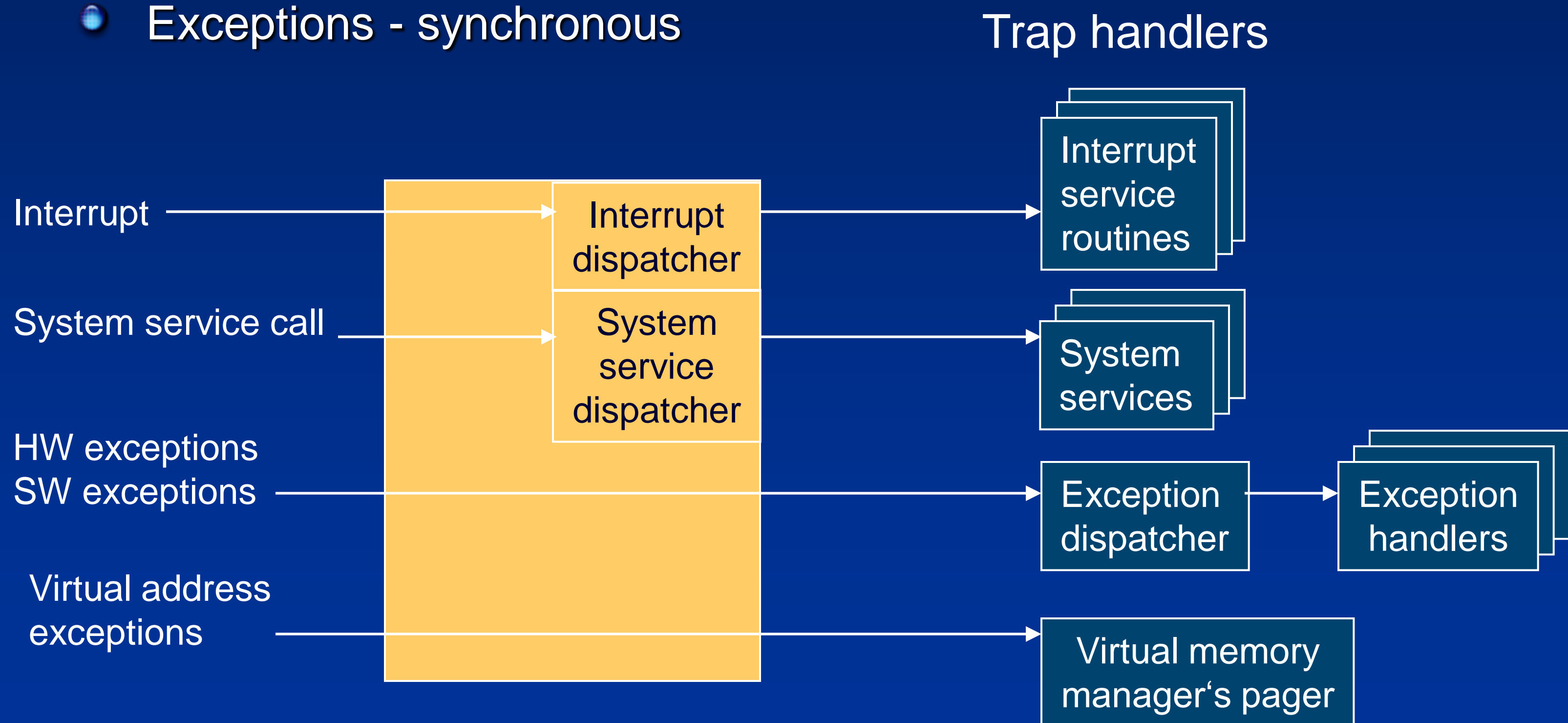
Code is run in kernel mode for one of three reasons:

1. Requests from user mode
   - Via the system service dispatch mechanism
   - Kernel-mode code runs in the context of the requesting thread
2. Interrupts from external devices
   - Windows interrupt dispatcher invokes the interrupt service routine
   - ISR runs in the context of the interrupted thread (so-called “arbitrary thread context”)
   - ISR often requests the execution of a “DPC routine,” which also runs in kernel mode
   - Time not charged to interrupted thread
3. Dedicated kernel-mode system threads
   - Some threads in the system stay in kernel mode at all times (mostly in the “System” process)
   - Scheduled, preempted, etc., like any other threads

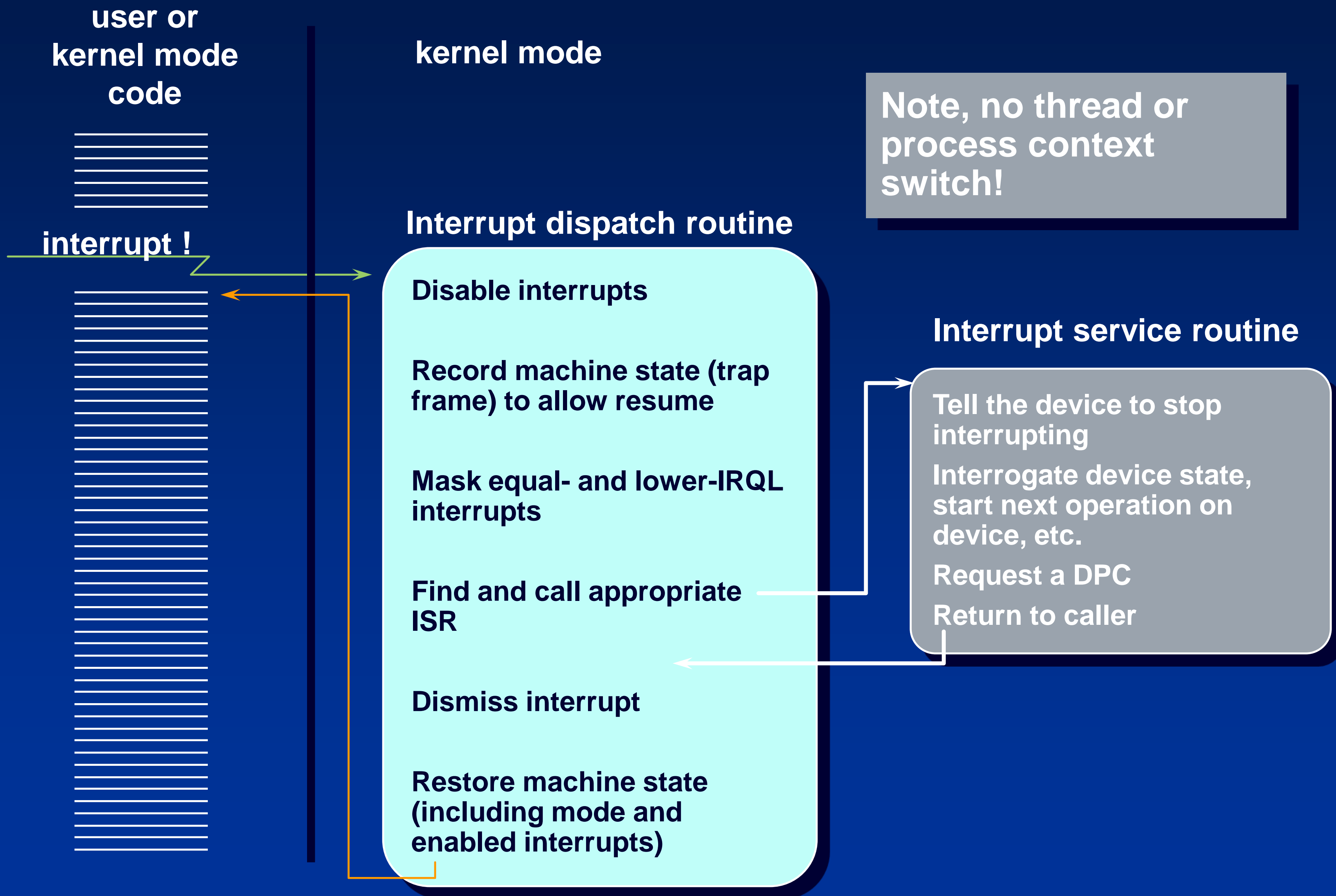
Trap Dispatching

Trap: processor’s mechanism to capture executing thread

- Switch from user to kernel mode
- Interrupts – asynchronous
- Exceptions - synchronous



Interrupt Dispatching



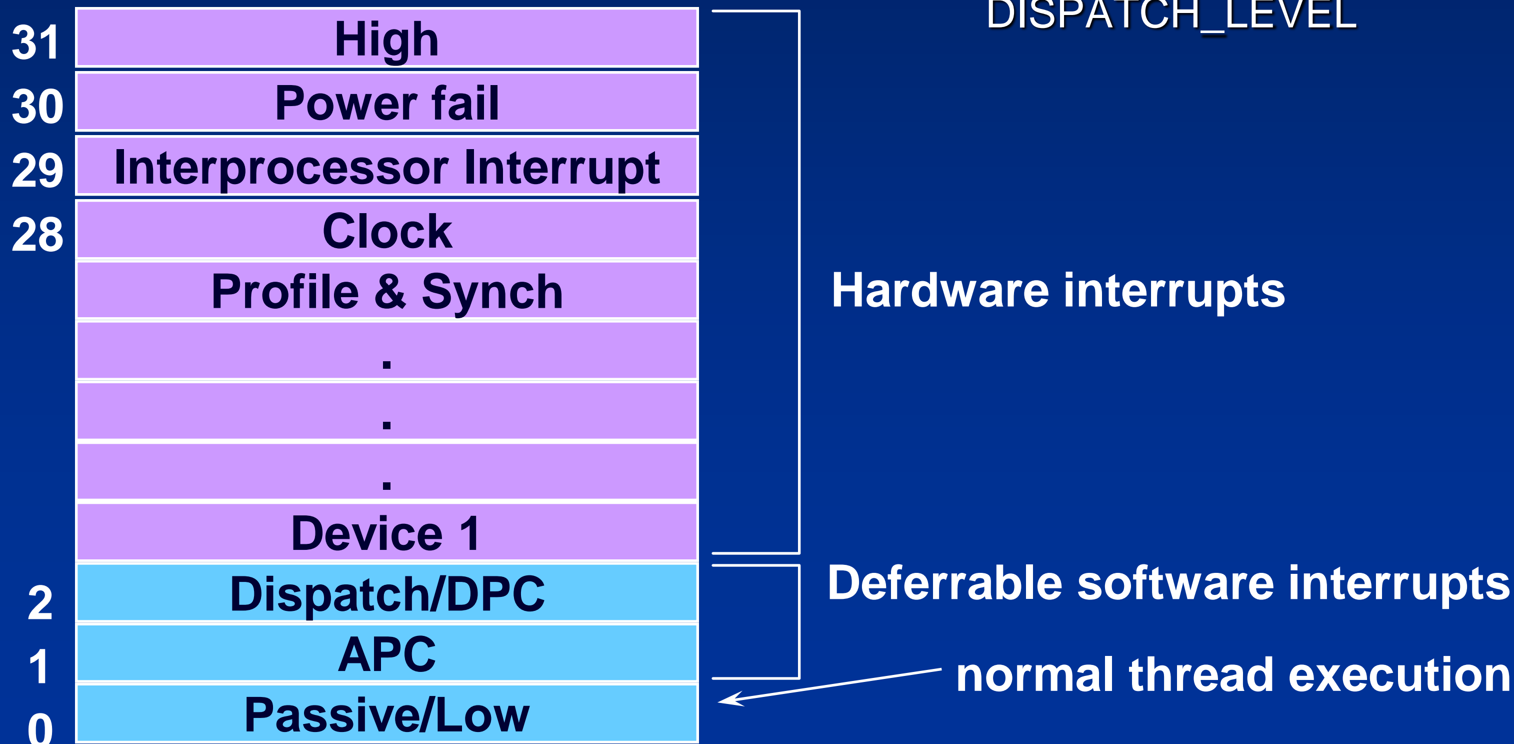
Hardware interrupt processing

- Interrupt dispatch table (IDT)
  - Links to interrupt service routines
- x86:
  - I/O interrupts come into one of the lines of interrupt controller
  - Interrupt controller interrupts processor (single line)
  - Processor queries for interrupt vector; uses vector as index to IDT
- After ISR execution, IRQL is lowered to initial level

Interrupt Precedence via IRQLs (x86)

- **IRQL = Interrupt Request Level**
  - the "precedence" of the interrupt with respect to other interrupts
  - Different interrupt sources have different IRQLs
  - not the same as IRQ (interrupt requests)

- IRQL is also a state of the processor
- Servicing an interrupt raises processor IRQL to that interrupt's IRQL
  - this masks subsequent interrupts at equal and lower IRQLs
- User mode is limited to IRQL 0
- No waits or page faults at IRQL >= DISPATCH_LEVEL



Predefined IRQLs

- **High**
  - used when halting the system (via *KeBugCheck()*)
- **Power fail**
  - originated in the NT design document, but has never been used
- **Inter-processor interrupt**
  - used to request action from other processor (dispatching a thread, updating a processors TLB (translation lookaside buffer), system shutdown, system crash)
- **Clock**
  - Used to update system‘s clock, allocation of CPU time to threads

Predefined IRQLs (contd.)

- **Profile**
  - Used for kernel profiling (see Kernel profiler – Kernprof.exe, Res Kit)
- **Device**
  - Used to prioritize device interrupts
- **DPC/dispatch** and **APC**
  - Software interrupts that kernel and device drivers generate
- **Passive**
  - No interrupt level at all, normal thread execution
- *Restriction: code running at DPC+ levels must not wait for an object which results in a thread re-scheduling*

IRQLs on 64-bit Systems

IRQL	x64	IA64
15	High/Profile	High/Profile/Power
14	Interprocessor Interrupt/Power	Interprocessor Interrupt
13	Clock	Clock
12	Synch (Srv 2003)	Synch (MP only)
	Device n	Device n
	.	.
4	.	Device 1
3	Device 1	Correctable Machine Check
2	Dispatch/DPC	Dispatch/DPC & Synch (UP only)
1	APC	APC
0	Passive/Low	Passive/Low

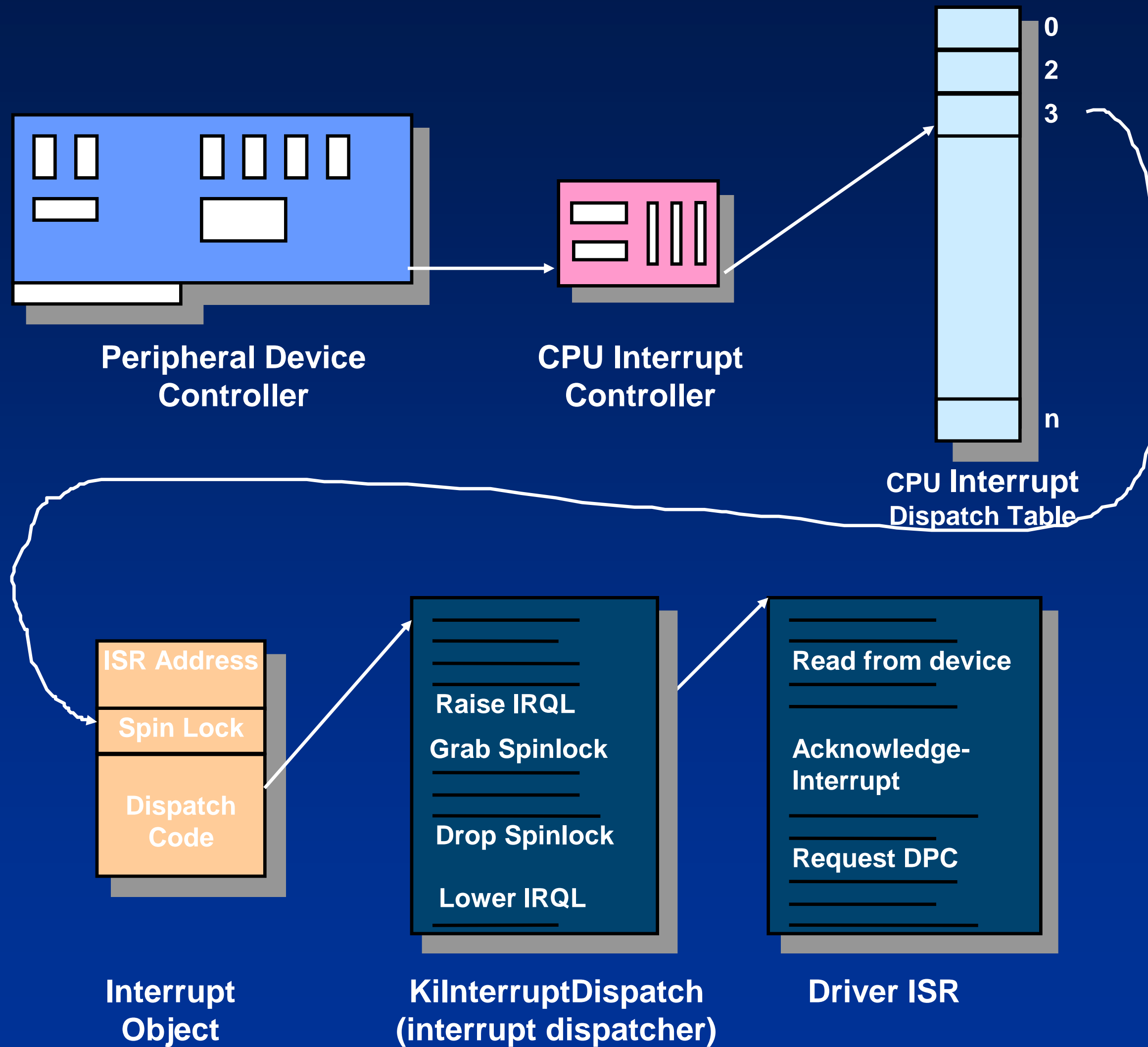
Interrupt Prioritization & Delivery

- IRQLs are determined as follows:
  - On x86, x64 & IA64 systems: IRQL = IDT vector number / 16
- On MP systems, which processor is chosen to deliver an interrupt?
  - By default, any processor can receive an interrupt from any device
    - Can be configured with IntFilter utility in Resource Kit
  - On x86 and x64 systems, the IOAPIC (I/O advanced programmable interrupt controller) is programmed to interrupt the processor running at the lowest IRQL
  - On IA64 systems, the SAPIC (streamlined advanced programmable interrupt controller) is configured to interrupt one processor for each interrupt source
    - Processors are assigned round robin for each interrupt vector

Interrupt object

- Allows device drivers to register ISRs for their devices
  - Contains dispatch code (initial handler)
  - Dispatch code calls ISR with interrupt object as parameter (HW cannot pass parameters to ISR)
- Connecting/disconnecting interrupt objects:
  - Dynamic association between ISR and IDT entry
  - Loadable device drivers (kernel modules)
  - Turn on/off ISR
- Interrupt objects can synchronize access to ISR data
  - Multiple instances of ISR may be active simultaneously (MP machine)
  - Multiple ISR may be connected with IRQL

Flow of Interrupts

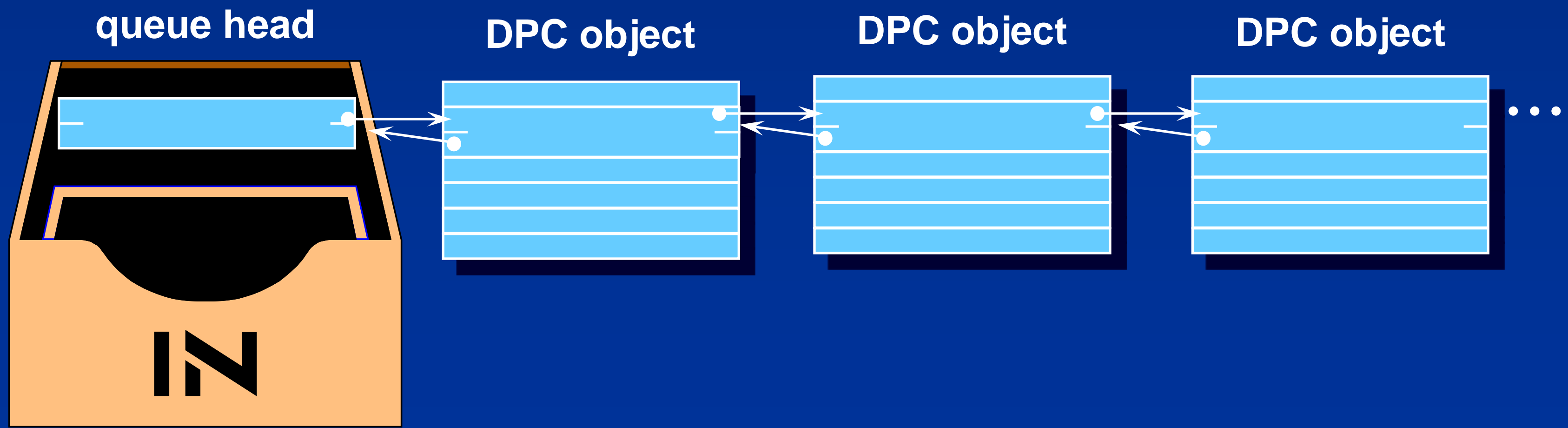


Software interrupts

- Initiating thread dispatching
  - DPC allow for scheduling actions when kernel is deep within many layers of code
  - Delayed scheduling decision, one DPC queue per processor
- Handling timer expiration
- Asynchronous execution of a procedure in context of a particular thread
- Support for asynchronous I/O operations

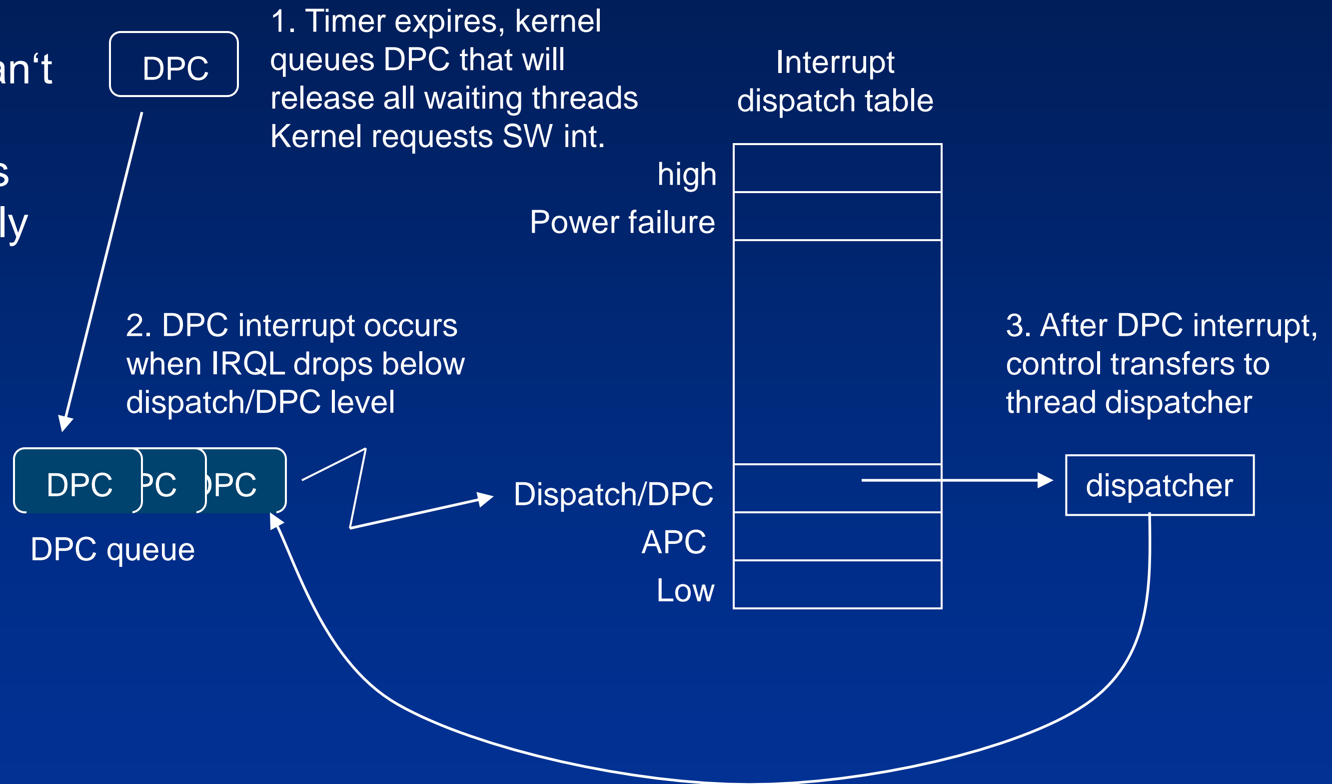
Deferred Procedure Calls (DPCs)

- Used to defer processing from higher (device) interrupt level to a lower (dispatch) level
  - Also used for quantum end and timer expiration
- Driver (usually ISR) queues request
  - One queue per CPU. DPCs are normally queued to the current processor, but can be targeted to other CPUs
  - Executes specified procedure at dispatch IRQL (or “dispatch level”, also “DPC level”) when all higher-IRQL work (interrupts) completed
  - Maximum times recommended: ISR: 25 usec, DPC: 100 usec
    - See http://msdn.microsoft.com/en-us/windows/hardware/gg487462.aspx



Delivering a DPC

DPC routines can't assume what process address space is currently mapped



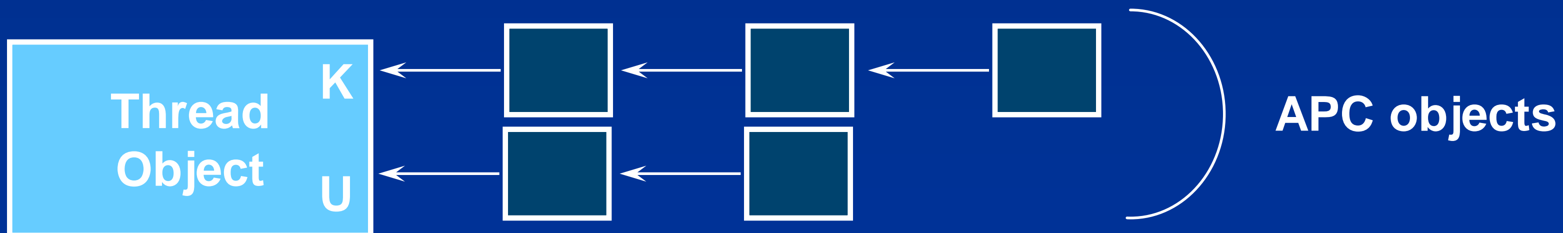
DPC routines can call kernel functions but can't call system services, generate page faults, or create or wait on objects

Asynchronous Procedure Calls (APCs)

- Execute code in context of a particular user thread
  - APC routines can acquire resources (objects), incur page faults, call system services
- APC queue is thread-specific
- User mode & kernel mode APCs
  - Permission required for user mode APCs
- Executive uses APCs to complete work in thread space
  - Wait for asynchronous I/O operation
  - Emulate delivery of POSIX signals
  - Make threads suspend/terminate itself (env. subsystems)
- APCs are delivered when thread is in alertable wait state
  - WaitForMultipleObjectsEx(), SleepEx()

Asynchronous Procedure Calls (APCs)

- Special kernel APCs
  - Run in kernel mode, at IRQL 1
  - Always deliverable unless thread is already at IRQL 1 or above
  - Used for I/O completion reporting from “arbitrary thread context”
  - Kernel-mode interface is linkable, but not documented
- “Ordinary” kernel APCs
  - Always deliverable if at IRQL 0, unless explicitly disabled (disable with KeEnterCriticalRegion)
- User mode APCs
  - Used for I/O completion callback routines (see ReadFileEx, WriteFileEx); also, QueueUserApc
  - Only deliverable when thread is in “alertable wait”

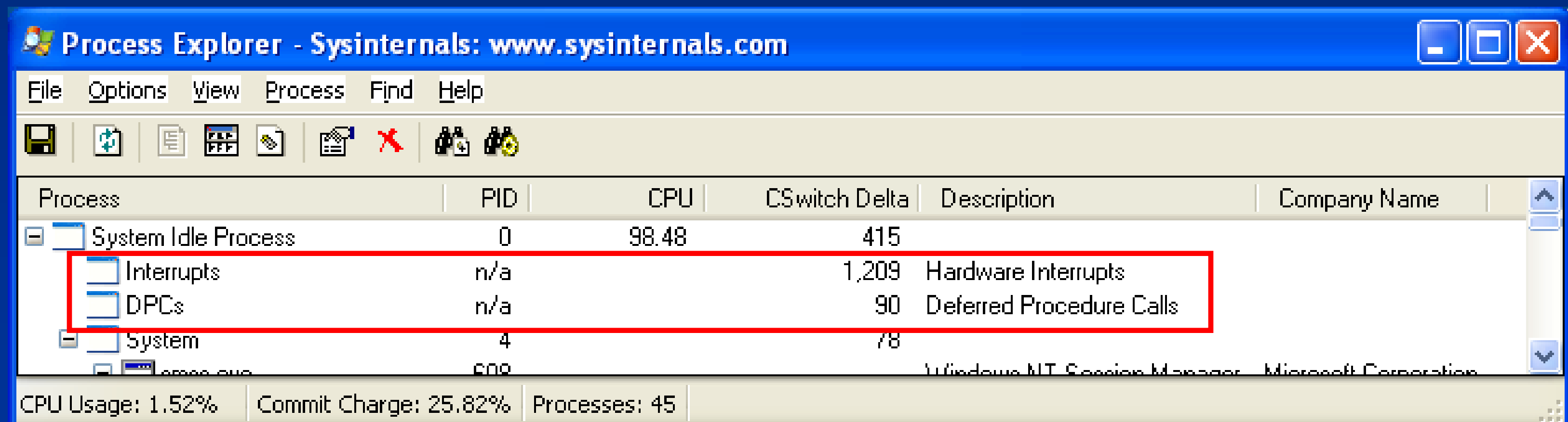


IRQLs and CPU Time Accounting

- Interval clock timer ISR keeps track of time
- Clock ISR time accounting:
  - If IRQL<2, charge to thread's user or kernel time
  - If IRQL=2 and processing a DPC, charge to DPC time
  - If IRQL=2 and not processing a DPC, charge to thread kernel time
  - If IRQL>2, charge to interrupt time
- Since the time servicing interrupts are NOT charged to interrupted thread, if system is busy but no process appears to be running, must be due to interrupt-related activity
  - Note: time at IRQL 2 or more ***is*** charged to the current thread's quantum (to be described)

Interrupt Time Accounting

- Task Manager includes interrupt and DPC time with the Idle process time
- Since interrupt activity is not charged to any thread or process, Process Explorer shows these as separate processes (not really processes)
  - Context switches for these are really number of interrupts and DPCs



Time Accounting Quirks

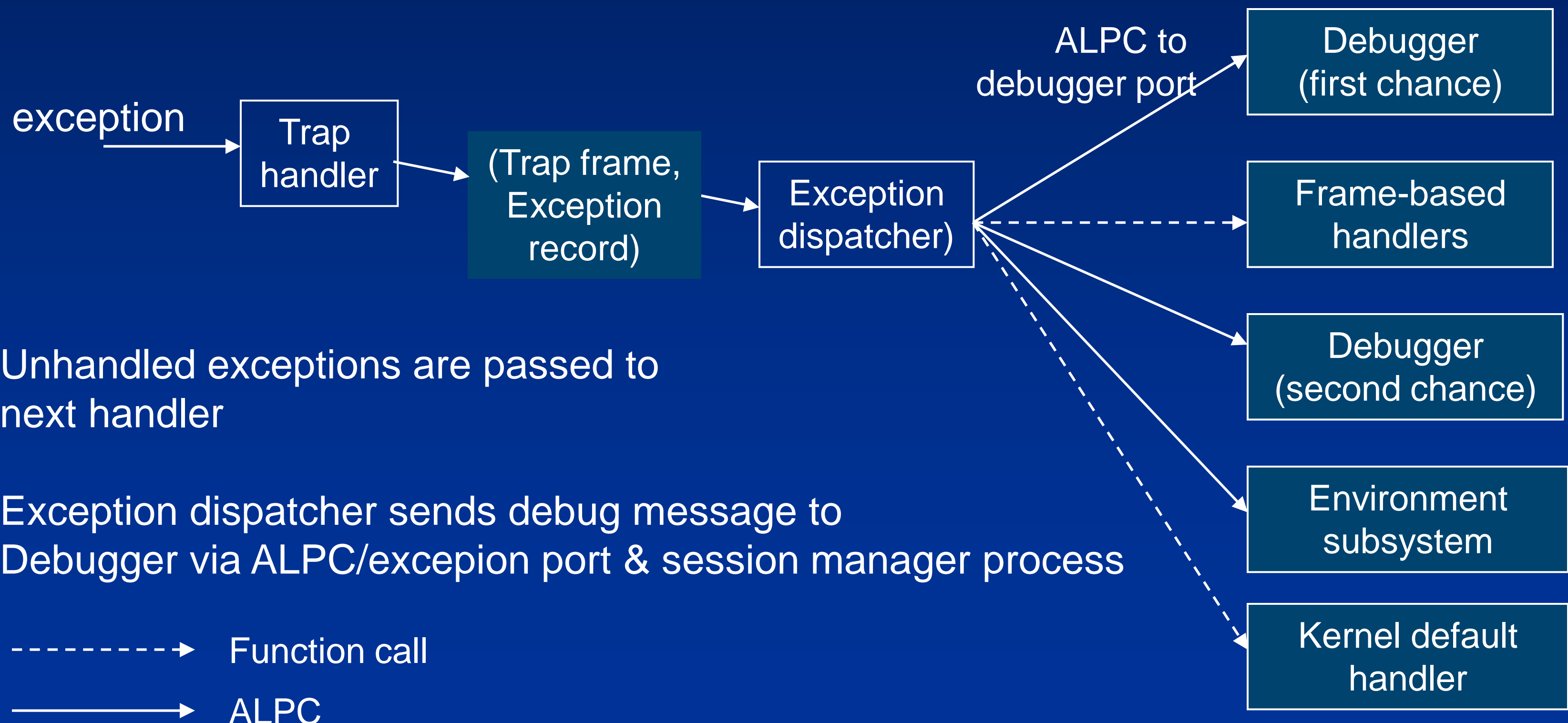
- Looking at total CPU time for each process may not reveal where system has spent its time
- CPU time accounting is driven by programmable interrupt timer
  - Normally 10 msec (15 msec on some MP Pentiums)
- Thread execution and context switches between clock intervals NOT accounted
  - E.g., one or more threads run and enter a wait state before clock fires
  - Thus threads may run but never get charged
- View context switch activity with Process Explorer
  - Add Context Switch Delta column

Exception Dispatching

- Exceptions are conditions that result directly from the execution of the program that is running
- Windows introduced a facility known as structured exception handling, which allows applications to gain control when exceptions occur
- The application can then fix the condition and return to the place the exception occurred,
  - unwind the stack (thus terminating execution of the subroutine that raised the exception), or
  - declare back to the system that the exception isn't recognized and the system should continue searching for an exception handler that might process the exception.

Exception Dispatching (contd.)

- Structured exception handling;
  - Accessible from MS VC++ language: __try, __except, __finally
  - See Jeffrey Richter, “Advanced Windows”, MS Press
  - See Johnson M.Hart, „Win32 System Programming“, Addison-Wesley



Internal Windows API exception handler

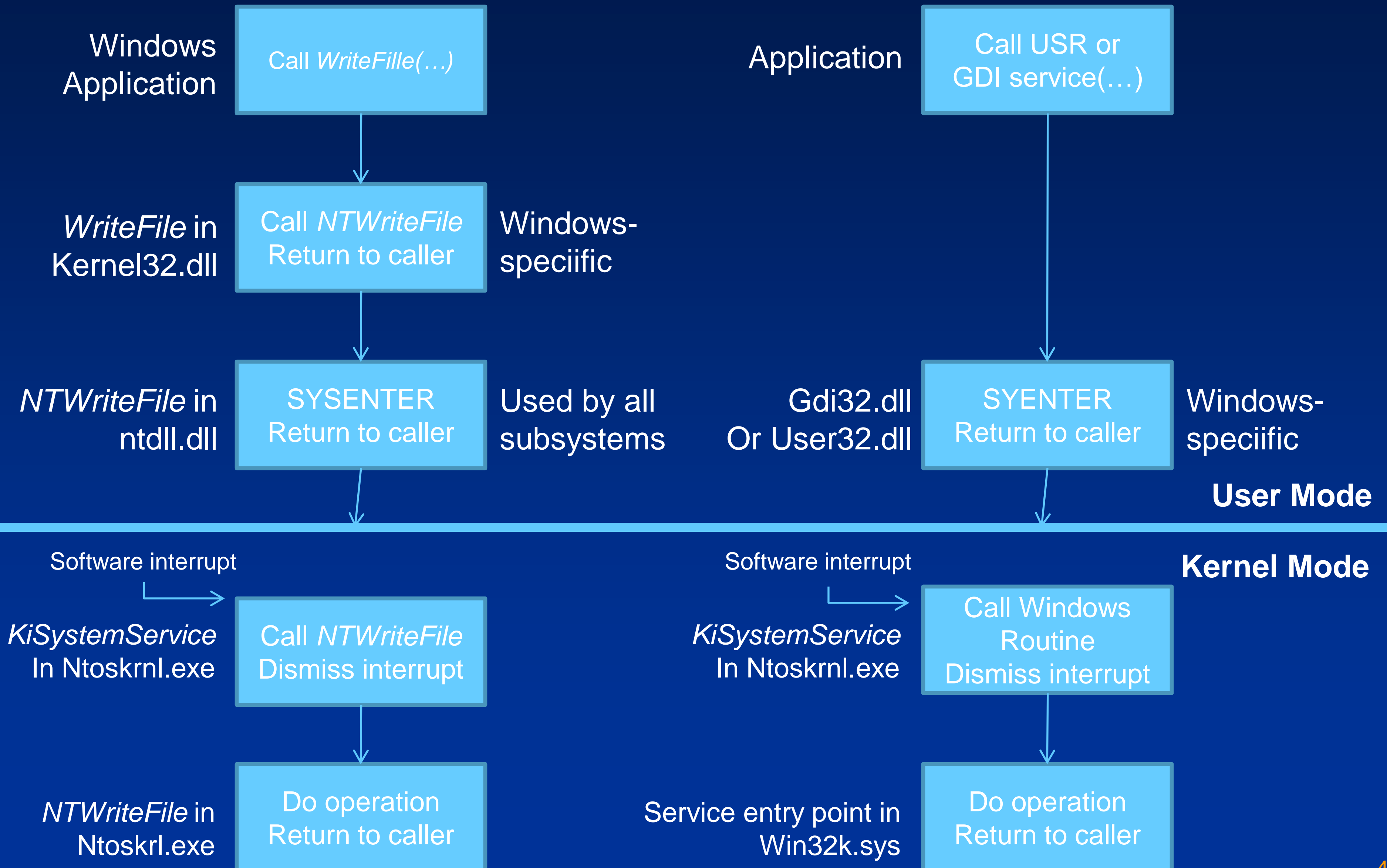
- Processes unhandled exceptions
  - At top of stack, declared in StartOfProcess()/StartOfThread()

```
void Win32StartOfProcess(LPTHREAD_START_ROUTINE lpStartAddr,
                         LPVOID lpvThreadParm) {
    __try {
        DWORD dwThreadExitCode = lpStartAddr(lpvThreadParm);
        ExitThread(dwThreadExitCode);
    }   __except(UnhandledExceptionFilter(
            GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
}
```

System Service Dispatching

- Triggered when executing an instruction assigned to system service dispatching
- Instruction depends on type of systems:
  - 32-bit system
    - Stores address to system service dispatch routine at *machine specific register* (MSR)
    - *sysenter* (*syscall* on AMD)
    - *sysexit* (*sysret* on AMD)
  - 64-bit system
    - Pass system call number in EAX register
    - *syscall*
    - *Sysret*
  - Kernel-mode system service dispatching
    - Drivers uses zwxxx system calls
    - Build fake interrupt stack , call *KiSystemService* directly
    - Set *previous mode* to kernel

System Service Dispatching



Lab: Interrupt Dispatching

- View IDT
- View the IRQL
- Use Kernel Profiler to Profile Execution
- Examine interrupt internals
- Monitor Interrupt and DPC activity

Lab: DPC

- Use Perfmon to check the following:
- Interrupts/sec,
- %Interrupt time,
- %DPC time,
- other DPC counters

Further Reading

- Mark E. Russinovich, *et al.* Windows Internals, 5th Edition, Microsoft Press, 2005.
- Chapter 3 - System Mechanisms
  - Trap Dispatching (from pp. 85)

Source Code References

- Windows Research Kernel sources
  - \base\ntos\ke\i386 (similar files for amd64)
    - Trap.asm, Trapc.c – Trap dispatcher
    - Spinlock.asm – Spinlocks
    - Clockint.asm – Clock Interrupt Handler
    - Int.asm, Intobj.c, Intsup.asm – Interrupt Processing
  - \base\ntos\ke
    - eventobj.c - Event object
    - mutntobj.c – Mutex object
    - semphobj.c – Semaphore object
    - timerobj.c, timersup.c – Timers
    - wait.c, waitsup.c – Wait support
  - \base\ntos\inc\ke.h – structure/type definitions