Windows Kernel Internals
Synchronization Mechanisms

David B. Probert, Ph.D.
Windows Kernel Development
Microsoft Corporation

Kernel synchronization mechanisms

Pushlocks
Fastref
Rundown protection
Spinlocks
Queued spinlocks
IPI
SLISTs

DISPATCHER_HEADER
KQUEUEs
KEVENTs
Guarded mutexes
Mutants
Semaphores
EventPairs
ERESOURCEs
Critical Sections

Push Locks

- Acquired shared or exclusive
- NOT recursive
- Locks granted in order of arrival
- Fast non-contended / Slow contended
- Sizeof(pushlock) == Sizeof(void*)
- Pageable
- Acquire/release are lock-free
- Contended case blocks using local stack

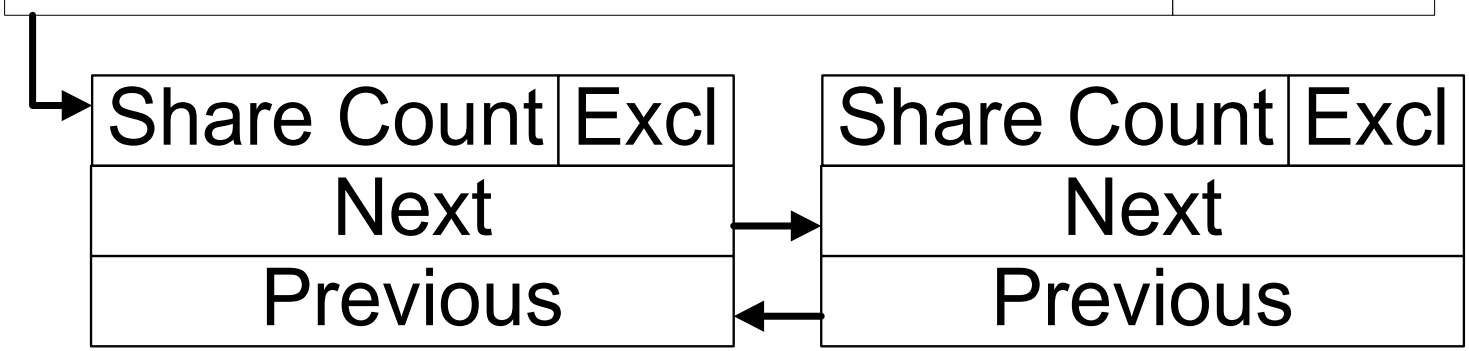
Pushlock format

Normal case

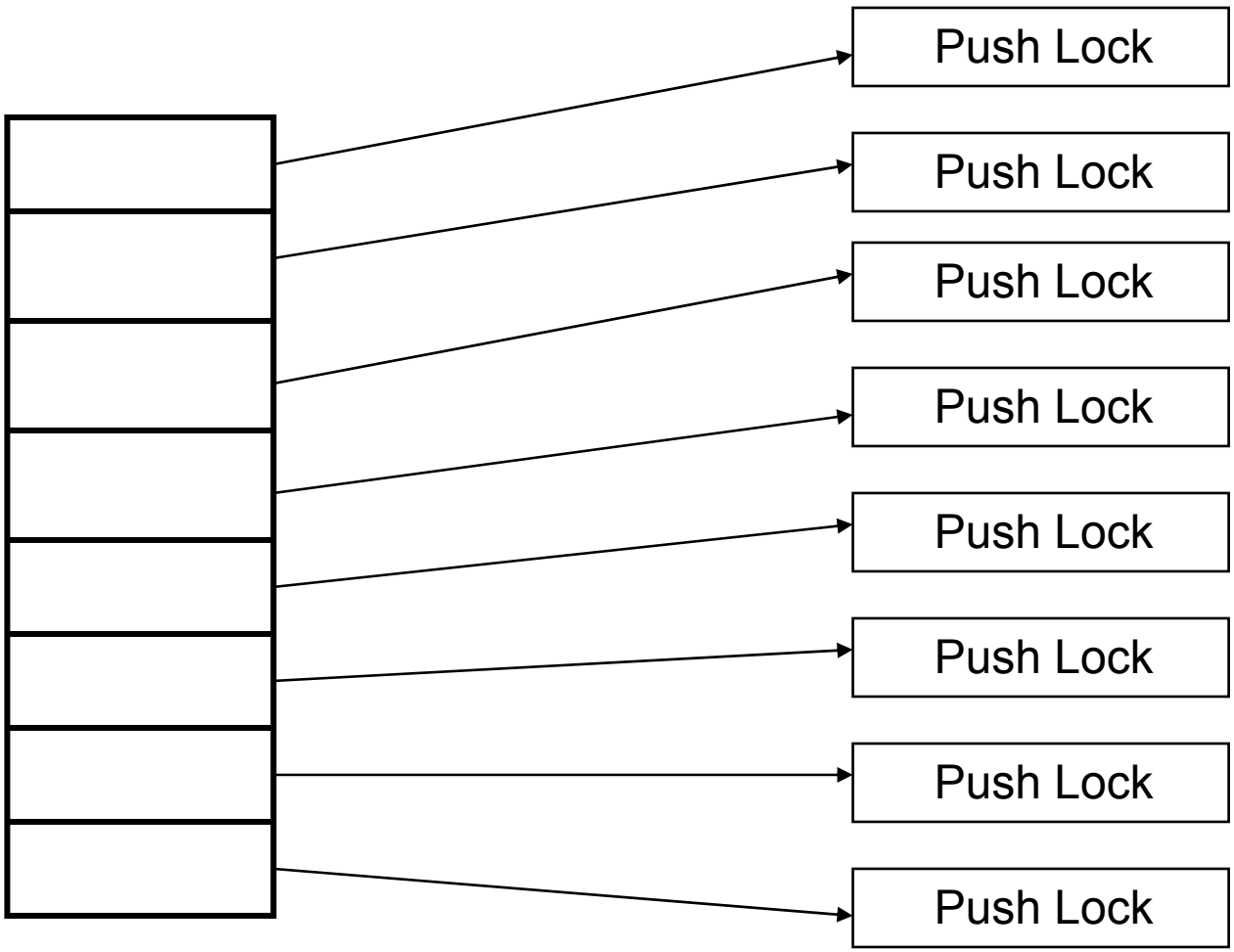
Share Count	Excl	W = 0

Contended case

Ptr to stack-local waitblock chain	W = 1



Cache Aware Push Locks



Pushlock non-contended cases

- Exclusive acquire:

  (SC=**0**,E=**0**,W=**0**) $\Rightarrow$ (SC=**0**,E=**1**,W=**0**)
- Exclusive release:

  (SC=**0**,E=**1**,W=**0**) $\Rightarrow$ (SC=**0**,E=**0**,W=**0**)
- Shared acquire:

  (SC=**n**,E=**0**,W=**0**) $\Rightarrow$ (SC=**n+1**,E=**0**,W=**0**)
- Shared release:

  (SC=**n+1**,E=**0**,W=**0**) $\Rightarrow$ (SC=**n**,E=**0**,W=**0**)

Pushlock contended cases

- Exclusive acquires:

  (SC=**0**,E=**1**,W=**0**) $\Rightarrow$ (P=wb(ssc=**0**,e=**1**),W=**1**)

  (SC=**n**,E=**0**,W=**0**) $\Rightarrow$ (P=wb(ssc=**n**,e=**1**),W=**1**)
- Shared acquire:

  (SC=**0**,E=**1**,W=**0**) $\Rightarrow$ (P=wb(ssc=**0**,e=**0**),W=**1**)

wb is a stack-allocated waitblock

ssc and e are the saved shared count and exclusive bit saved in the wb

Pushlock contended cases

Shared releasing threads:

- Search wb list for a wb' with ssc>0 (or e==1)
- If (InterlockedDecrement(wb'.ssc)) == 0) *fall through to the exclusive case*
- note that multiple threads may release but only one will decrement to 0

ExfAcquirePushLockExclusive

while(1)

if PushLock FREE, confirm by setting to Exclusive DONE

if PushLock.Waiting
  set WB (SSC=0, E=1, next=PushLock.Next)
else
  n = PushLock.ShareCount
  set WB (SSC=n, E=1, next=NULL)

Attempt to set Pushlock.Next = WB, Pushlock.Waiting = 1
Loop on failure
Wait for event

ExfAcquirePushLockShared

```
while(1)
   if PushLock FREE or shared (no waiters) count++.  DONE

   if Pushlock.Exclusive OR Pushlock.Waiting   // E or SSC
      if Pushlock.Waiting
         set WB(SSC=0, E=0, Next=PushLock.Next)
      else
         set WB(SSC=0, E=0, Next=NULL)
   Attempt to set Pushlock.Next = WB, Pushlock.Waiting = 1
   Loop on failure
   Wait for event
```

Pushlock contended cases

Exclusive releasing threads:

- Search wb list for:
  - continuous chain of wb with ssc > 0
  - or, a *wb'* with e == 1
- Can then split the list one of two ways
  - the list and gives away either s or e

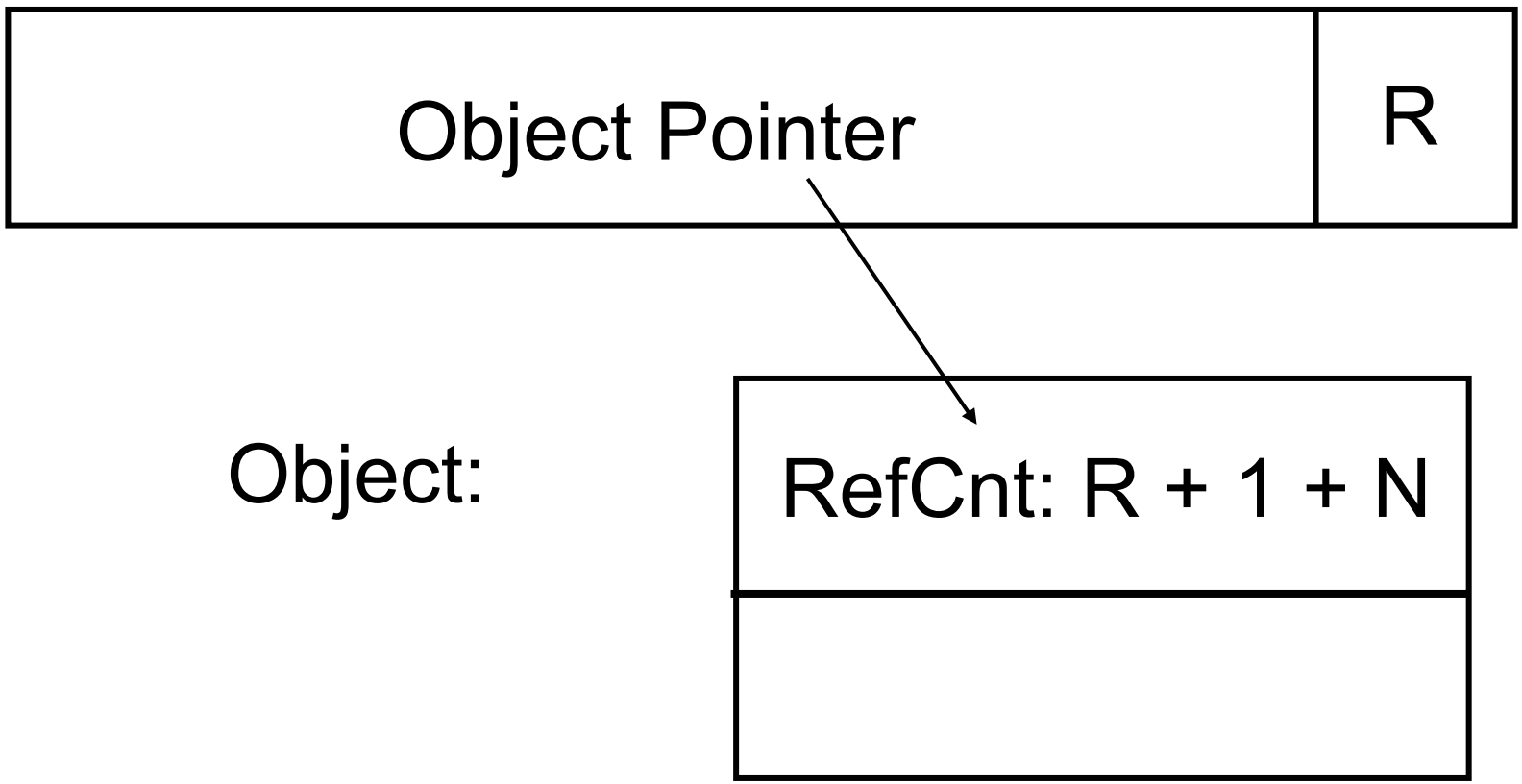
Fast Referencing

- Used to protect rarely changing reference counted data
- Small pageable structure that's the size of a pointer
- Scalable since it requires no lock acquires in over 99% of calls

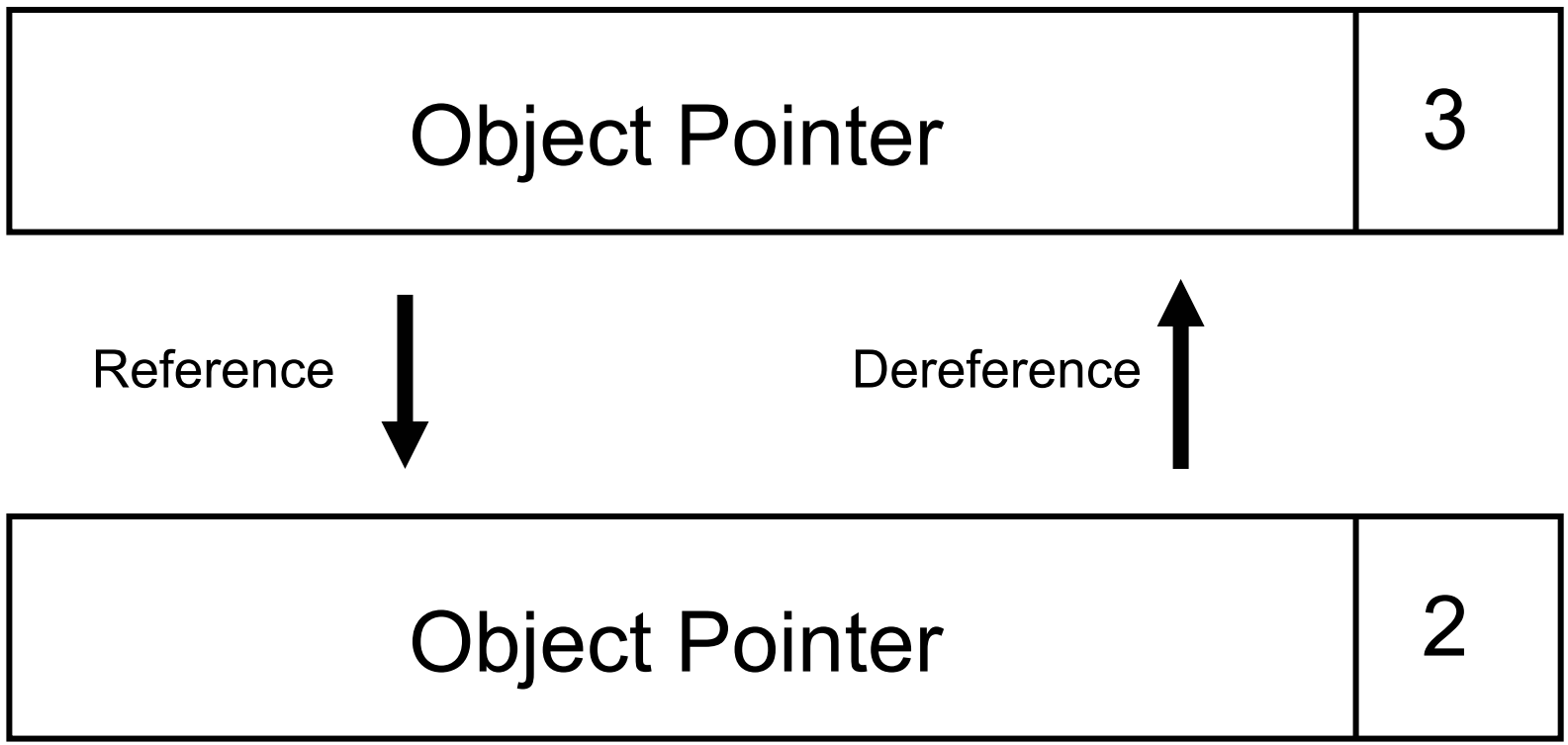
Fast Referencing Example

```
// Get a reference to the token using the fast path if we can
Token = ObFastReferenceObject (&Process->Token);
    if (Token == NULL) {
    // The fast path failed so we have to obtain the lock first
     PspLockProcessSecurityShared (Process);
       Token = ObFastReferenceObjectLocked
                                          (&Process->Token);
       PspUnlockProcessSecurityShared (Process);
    }
```

Fast Referencing Internals



Obtaining a Fast Reference

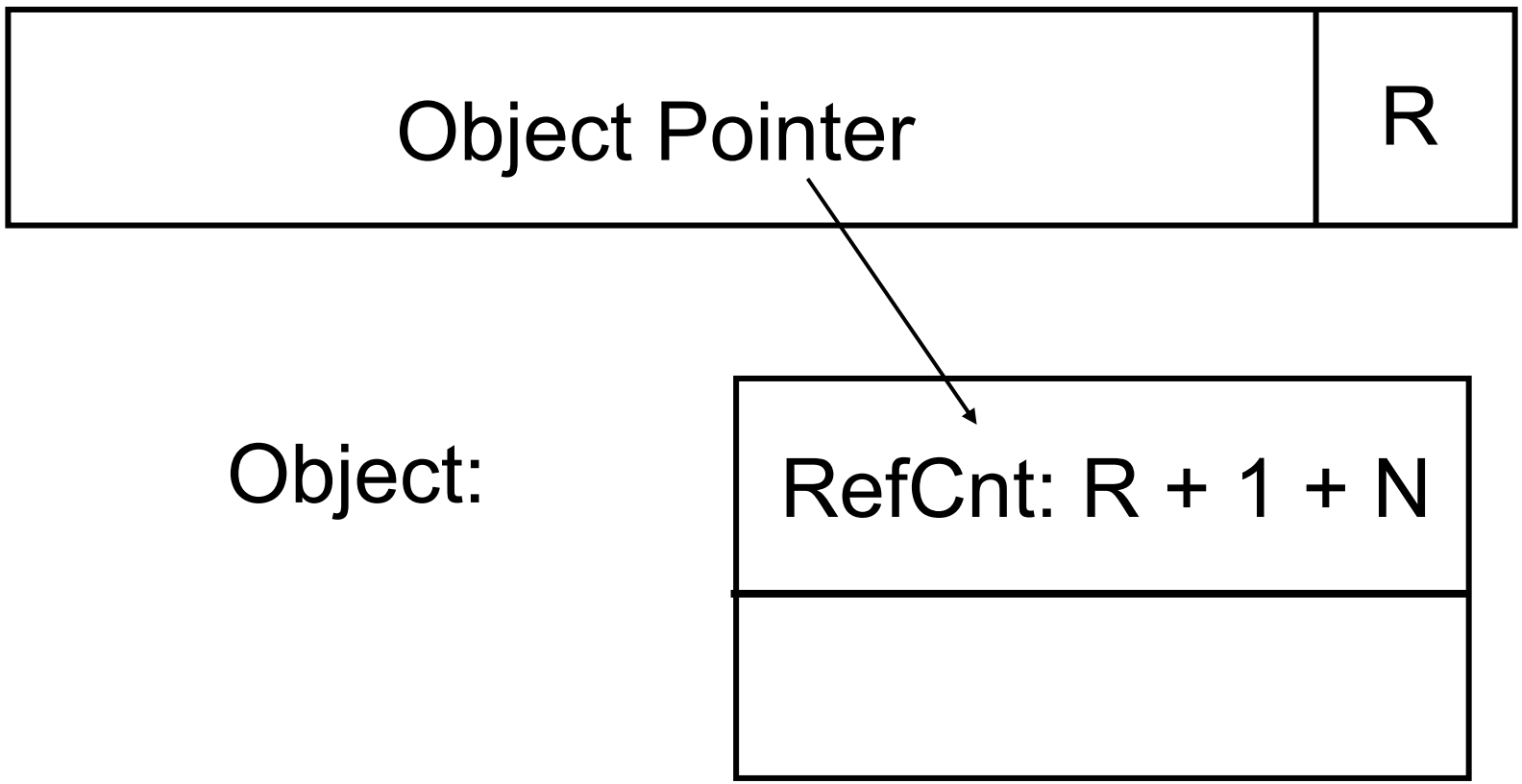


Replacing Fast Referenced Objects

```
// Swap new token for old
OldToken = ObFastReplaceObject
                        (&Process->Token, NewToken);

// Force any threads out of the slow ref path
PspLockProcessSecurityExclusive (Process);
PspUnlockProcessSecurityExclusive (Process);
```

Fast Referencing Internals



Rundown Protection

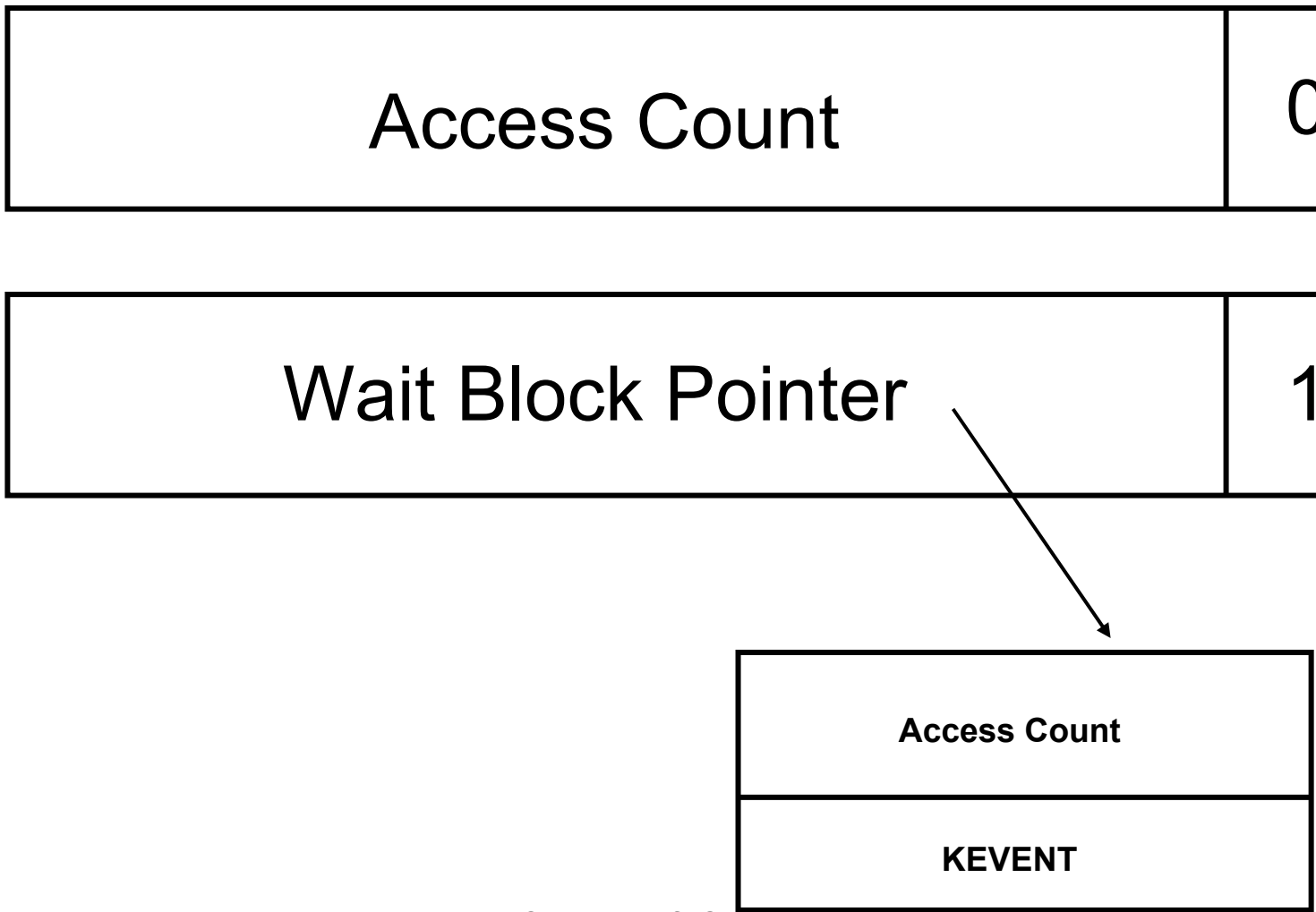
- Protects structures that last a long time but are eventually rundown (destroyed)
- Small (The size of a pointer) and can be used in pageable data structures
- Acquire and release are fast and lock free in the non-rundown case
- Rundown protection can be reset but we don’t use that currently

Rundown Protection Example

```
if (ExAcquireRundownProtection (&Parent->RundownProtect)) {
      SectionObject = Parent->SectionObject;
      if (SectionObject != NULL) {
         ObReferenceObject (SectionObject);
      }
      ExReleaseRundownProtection (&Parent->RundownProtect);
  }

  if (SectionObject == NULL) {
      Status = STATUS_PROCESS_IS_TERMINATING;
      goto exit_and_deref;
  }
```

Rundown Protection Internals



Spinlocks

Spinlock Acquire:

```
A:  lock bts    dword ptr [LockAddress], 0
    jc          done
S:  test        dword ptr [LockAddress], 1
    jz          A
    pause
    jmp         S
```

Spinlock Release:

```
    lock and    byte ptr [LockAddress], 0
```

Queued Spinlocks

To acquire, processor queues to lock

At release, lock passes to queued processor

Waiting processors spin on local flag

Advantages:

- Reduced coherency traffic
- FIFO queuing of waiters

Kernel Queued Lock use

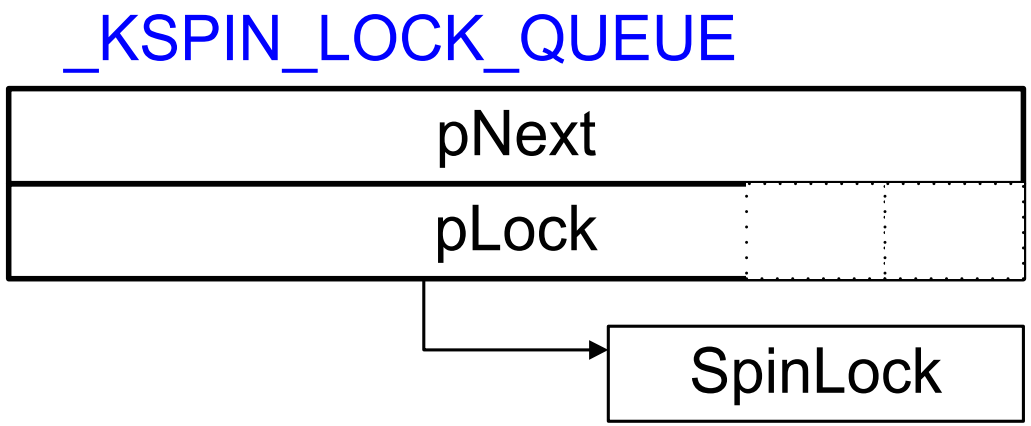
DispatcherLock
PfnLock
SystemSpaceLock
VacbLock
MasterLock
NonPagedPoolLock
IoCancelLock
WorkQueueLock
IoVpbLock

IoDatabaseLock
IoCompletionLock
NtfsStructLock
AfdWorkQueueLock
BcbLock
MmNonPagedPoolLock

Kept per-processor:
PRCB->LockQueue[]

e.g. for each processor's control block:
Prcb->LockQueue[idxDispatcherLk].Next = NULL;
Prcb->LockQueue[idxDispatcherLk].Lock = &KiDispatcherLk

Queued Spinlocks



KiAcquireQueuedLock(pQL)

```
prev = Exch (pQL->pLock, pQL)
if (prev == NULL)
    pQL->pLock |= LQ_OWN
    return
pQL->pLock |= LQ_WAIT
prev->pNext = pQL
while (pQL->pLock & LQ_WAIT)
   KeYieldProcessor()
return
```

KiReleaseQueuedLock(pQL)

```
pQL->pLock &= ~(LQOWN | LQWAIT)
lockval = *pQL->pLock
if (lockval == pQL)
    lockval = CmpExch(pQL->pLock, NULL, pQL)
if (lockval == pQL) return

while (! (pWaiter = pQL->pNext))
   KeYieldProcessor()
pWaiter->pLock ^= (LQOWN|LQWAIT)
pQL->pNext = NULL
return
```

QueuedLock examples

Action	spinlk	QL-P0	QL-P1	QL-P2
Initial ⇒	null	null-	null-	null, -
A-P0 ⇒	QL0	null-own	null-	null, -
A-P1 ⇒	QL1	QL1-own	null-wait	null, -
A-P2 ⇒	QL2	QL1-own	QL2-wait	null-wait
R-P0 ⇒	QL2	null-	QL2-own	null-wait
R-P1 ⇒	QL2	null-	null-	null-own
R-P2 ⇒	null	null-	null-	null-

InterProcessor Interrupts (IPIs)

Synchronously execute a function on every processor

```
KeIpiGenericCall(fcn, arg)
    oldIrql = RaiseIrql(SYNCH_LEVEL)
    Acquire(KiReverseStallIpiLock)
    Count = KeNumberOfProcessors()
    KiIpiSendPacket(targetprocs, fcn, arg, &count)
    while (count != 1) KeYieldProcessor()
    RaiseIrql(IPI_LEVEL)
    Count = 0          // all processors will now proceed
    fcn(arg)
    KiIpiStallOnPacketTargets(targetprocs)
    Release(KiReverseStallIpiLock)
    LowerIrql(OldIrql)
```

KiIpiSendPacket

```
me = PCR->PRCB;
me->PbTargetSet = targetset
me->PbWorkerRoutine = fcn
me->PPbCurrentPacket = arg
for each p in targetset
    them = KiProcessorBlock[p]->PRCB
    while (CMPEXCH(them->PbSignalDone, me, 0)) YIELD
    HalRequestIpi(p)
return

// the IPI service routine will invoke KiIpiGenericaCallTarget
// on each processor with fcn and arg as parameters
```

KiIpiGenericCallTarget

InterlockedDecrement(Count)

while (count > 0) KeYieldProcessor()

fcn(arg)

KiIpiSignalPacketDone()

return

Interlocked Sequenced Lists

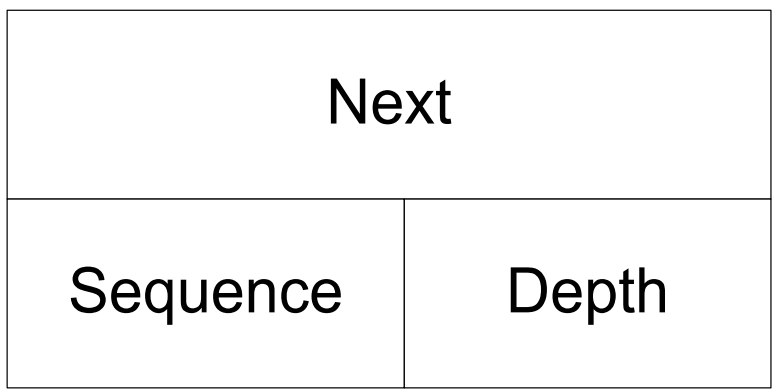
Singly linked lists

Header contains depth and a sequence no.

Allows for lock-free pushes/pops

Used primarily in memory/heap management

SLIST_HEAD



PushEntrySList(slh, s)

```
do { SLIST_HEAD nslh, oslh
    oslh->seqdepth = slh->seqdepth
    oslh->next = slh->next
    nslh->seqdepth = oslh->seqdepth + 0x10001
    nslh->next = s
    s->next = oslh->next
} until CmpExch64(slh, nslh, oslh) succeeds
return
```

s = PopEntrySList(slh)

```
do { SLIST_HEAD nslh, oslh, *s
    oslh->seqdepth = slh->seqdepth
    s = oslh->next = slh->next
    if (!s) return NULL
    nslh->seqdepth = oslh->seqdepth – 1  //depth--
    nslh->next = s->next   // can fault!
} until CmpExch64(slh, nslh, oslh) succeeds
return s
```

PopEntrySList faults

s->next may cause a fault:

- top entry allocated on another processor
- top entry freed between before *s* referenced

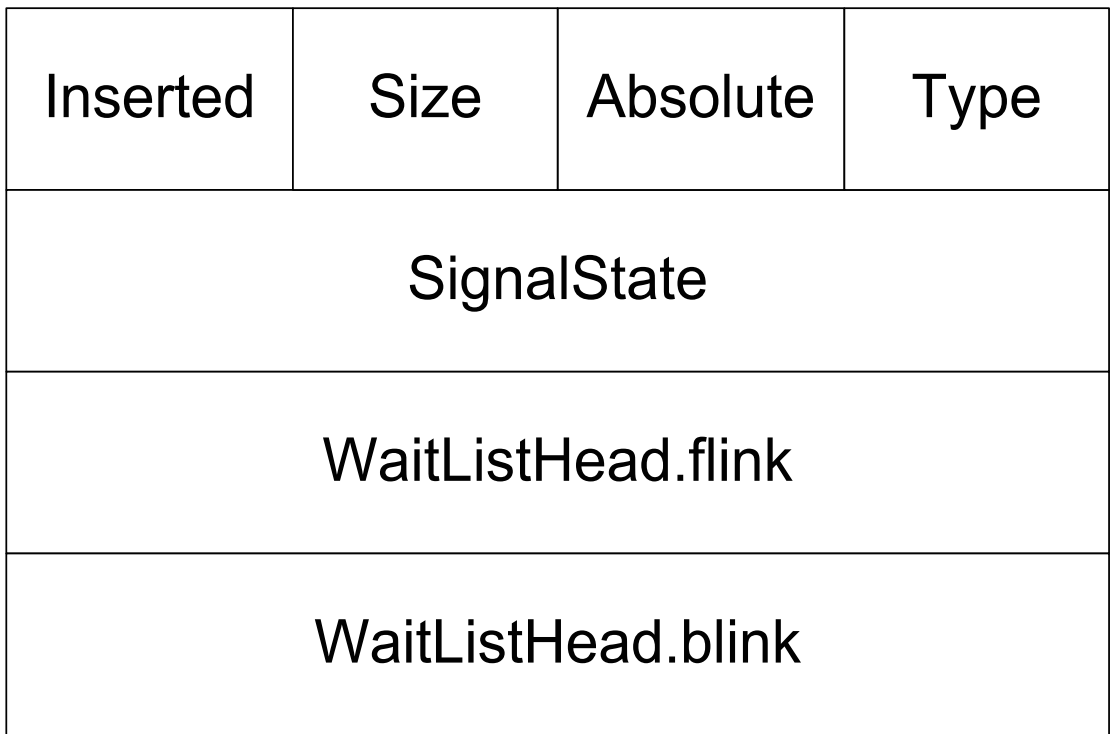
Access fault code special cases this:

- the faulting s->next reference is skipped
- the compare/exchange fails
- the pop is retried

DISPATCHER_HEADER

Fundamental kernel synchronization mechanism

Equivalent to a KEVENT



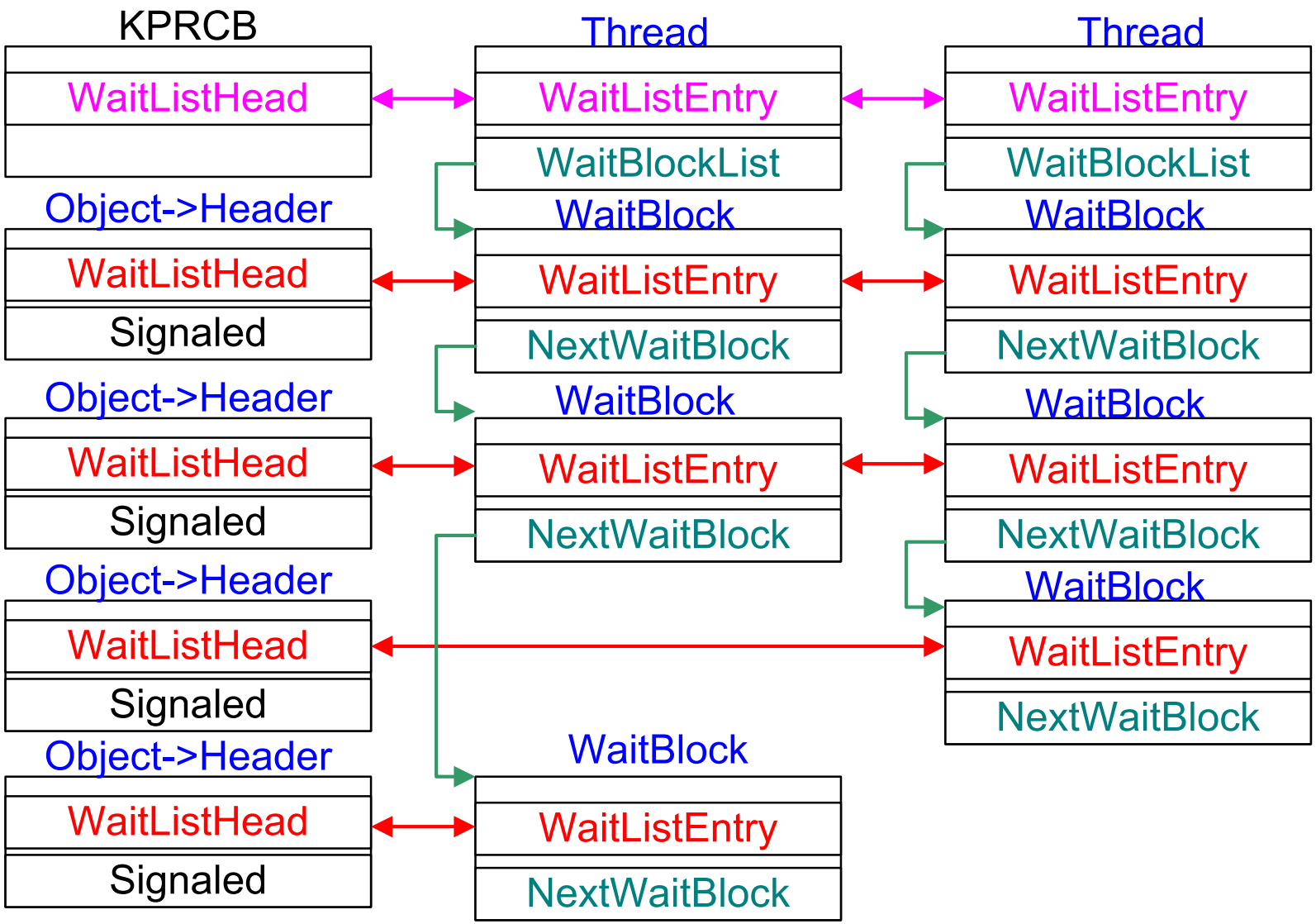
WAIT fields

KTHREAD WAIT fields

- LONG_PTR **WaitStatus**;
- PRKWAIT_BLOCK **WaitBlockList**;
- BOOLEAN **Alertable**, **WaitNext**;
- UCHAR **WaitReason**;
- LIST_ENTRY **WaitListEntry**; **// Prcb->WaitListHead**
- KWAIT_BLOCK **WaitBlock[4]**; **// 0, 1, event/sem, timer**

KWAIT_BLOCK – *represents waiting thread*

- LIST_ENTRY **WaitListEntry;** **// Object->WaitListHead**
- PKTHREAD **Thread**;
- PVOID **Object**;
- PKWAIT_BLOCK **NextWaitBlock**; **// Thread->WaitBlockList**
- USHORT **WaitKey**, **WaitType**;



WAIT notes

KPRCB WaitList used by balance manager for swapping out stacks

Several WaitBlocks available in thread itself to save allocation time

Waitable objects all begin with standard dispatcher header

KeSetEvent (KEVENT Event)

```
// while holding DispatcherDatabaseLock
old = Event->SignalState
Event->SignalState = 1
if old == 0 && !Empty(Event->WaitList)
    if Event->Type == Notification
        wakeup every thread in the WaitList
    else // Synchronization type
        wakeup one thread in the WaitList
```

Kernel Queue Object (KQUEUE)

Mechanism for thread work queues

Used by worker threads and I/O completion ports

KQUEUE Operations:

- KeInitializeQueue(pQ, MaxThreads)
- KeInsertQueue(pQ, pEntry)
- KeInsertHeadQueue(pQ, pEntry)
- KeRemoveQueue(pQ, WaitMode, Timeout)
- KeRundownQueue(pQ)

KQUEUE

DISPATCHER_HEADER
EntryListHead[2]
CurrentCount
MaximumCount
ThreadListHead[2]

Each _KTHREAD structure contains a field

KQUEUE Queue

for use with the kernel queue mechanism

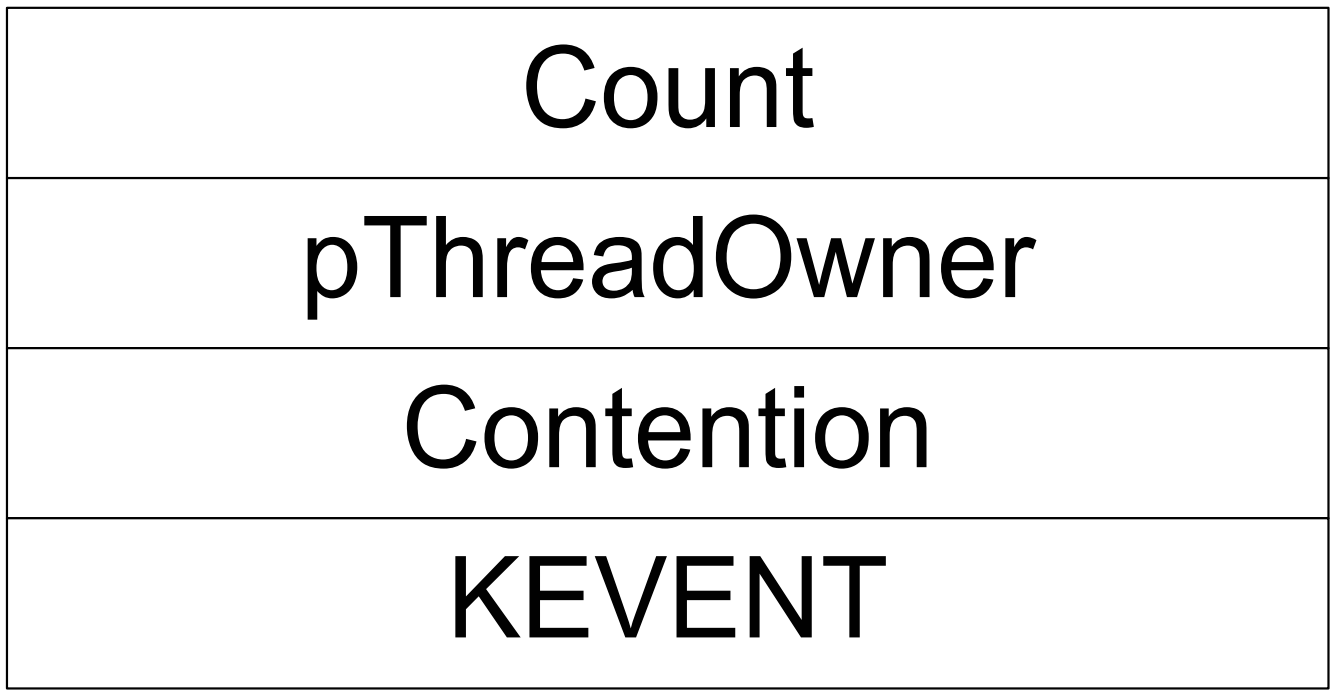
KiInsertQueue(pQ, pEntry, bHead)

```
// implements both KeInsert routines
// called holding the DispatcherDatabaseLock
oldState = pQ->Header.SignalState
if pQ->Header points to a waiting thread
        && CurrentThreads < MaximumThreads
        && the current thread isn't waiting on this queue
    Call KiReadyThread(lastThreadQueued)
else
    pQ->Header.SignalState++
    Insert pEntry in pQ->EntryListHead according to bHead
return oldState
```

entry = KeRemoveQueue(pQ)

```
// holding the DispatcherDatabaseLock
if (oldQ = thread->Queue) != pQ
    if (oldQ)
        remove thread from oldQ
        try to unwait a thread from oldQ->Header's waitlist
    insert thread in pQ->ThreadListHead
else
    pQ->CurrentCount --
while (1)
    if the queue isn't empty and Current < Max
        remove Entry from head of EntryListHead
        start a waiting thread
        return Entry
    else
        queue current thread to pQ->Header
```

Guarded Mutexes



Guarded Mutexes

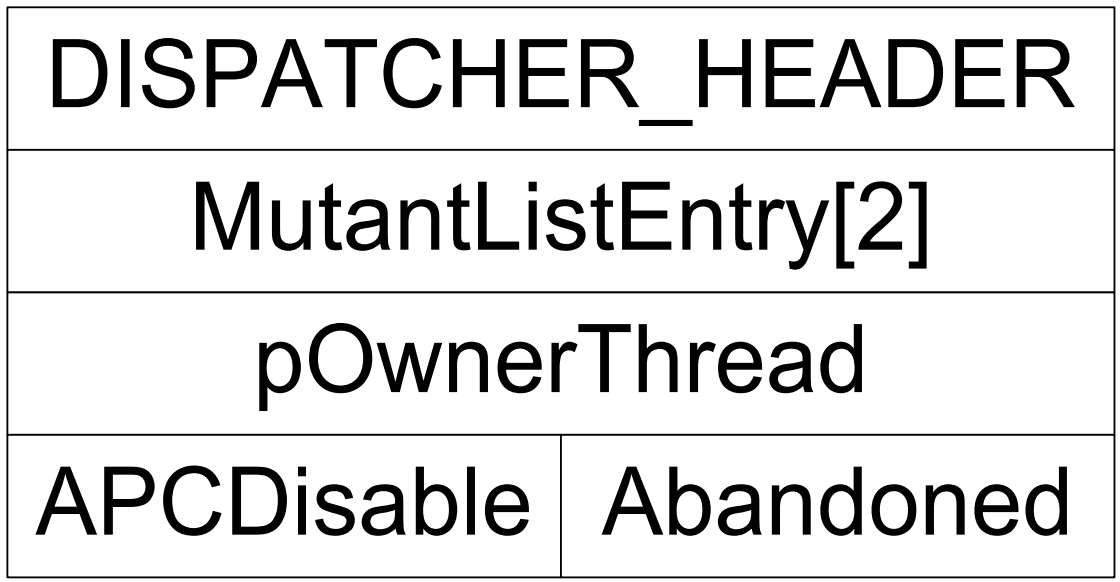
KeAcquireGuardedMutex()

```
KeEnterGuardedRegionThread(Thread)
if (InterlockedDecrement (&Mutex->Count) != 0)
    Mutex->Contention += 1
    KeWaitForSingleObject(&Mutex->Event)
Mutex->Owner = Thread
```

KeReleaseGuardedMutex()

```
Mutex->Owner = NULL
if (InterlockedIncrement (&Mutex->Count) <= 0)
    KeSetEventBoostPriority(&Mutex->Event)
KeLeaveGuardedRegionThread(Thread)
```

KMUTANT

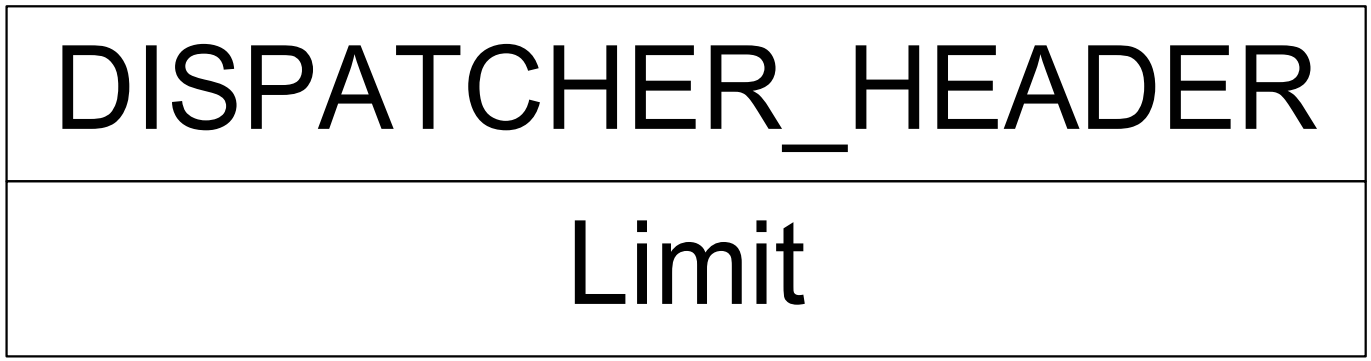


Exposed to usermode via CreateMutex

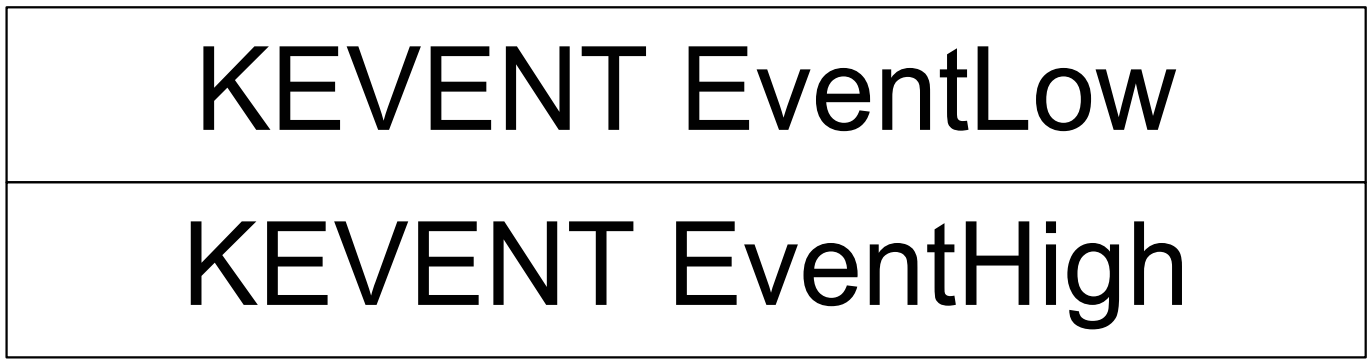
Differences from KEVENT:

Checks ownership and can be abandoned

KSEMAPHORE



KEVENTPAIRS



KiSetServerWaitClientEvent()

- sets the specified server event
- waits on specified client event
- wait is performed such that an optimal switch to the waiting thread occurs if possible

ERESOURCE
kernel reader/writer lock

SystemListEntry[2]	
pOwnerTable	
Flag	ActiveCount
pShWaitersSemaphore	
pExWaitersEvent	
pOwnerThreads[2]	
ContentionCount	
nExWaiters	nShWaiters

User-mode Critical Sections

Primary Win32 locking primitive

Backed by kernel semaphore *as-needed*

- i.e. in non-contended case no semaphore!

Because semaphore allocated *on-demand*

- acquiring/releasing critical section can fail
- fixed in Windows Server 2003
- under low memory replaces with keyed event

CRITICAL_SECTION

pDebugInfo
LockCount
RecursionCount
OwningThreadHandle
LockSemaphoreHandle
SpinCount

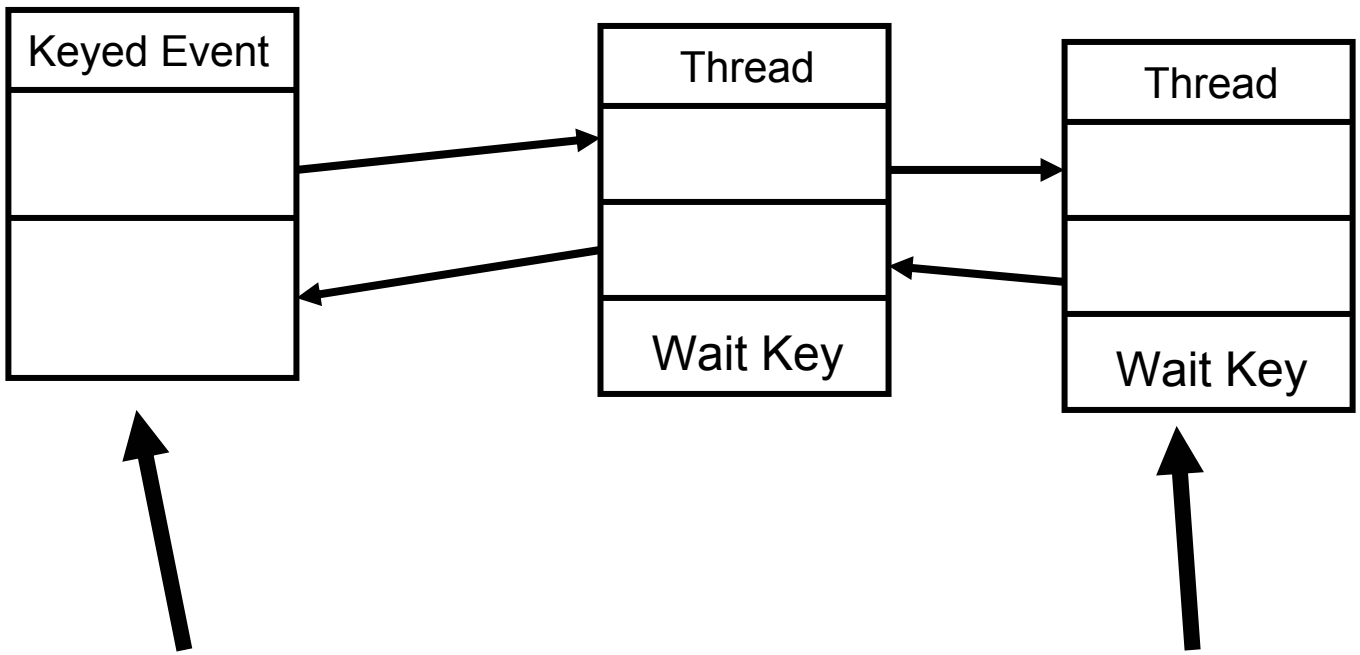
CRITICAL_SECTION_DEBUG

pCriticalSection
ProcessLocksEntry[2]
EntryCount
ContentionCount
LockSemaphoreHandle
SpinCount

Keyed Events

- These are a way to solve the raising Enter/LeaveCriticalSection problem
- Require no storage allocation so they can't fail
- Reuse existing LPC structures so don't require additions to the thread
- Don't prevent the kernel stack from being swapped like push locks etc do

Keyed Event Internals



Only one of these
in the entire system

Wait key is the Critical
section address

Summary

Pushlocks
Fastref
Rundown protection
Spinlocks
Queued spinlocks
IPI
SLISTs

DISPATCHER_HEADER
KQUEUEs
KEVENTs
Guarded mutexes
Mutants
Semaphores
EventPairs
ERESOURCEs
Critical Sections

Discussion