Operating Systems

(1DT020 & 1TT802)

Lecture 8

Memory Management : Virtual memory mechanism

April 28, 2008

Léon Mugwaneza

http://www.it.uu.se/edu/course/homepage/os/vt08

Review : Basic Readers/Writers Solution

- **Correctness Constraints:**
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time

- **Monitor DataBase**
  - 4 external procedures :
    - » BeginRead, EndRead,
    - » BeginWrite, EndWrite
  - State variables
    - » int AR: # active readers; initially = 0
    - » int WR: # waiting readers;
    - » initially = 0
    - » int AW: # active writers; initially = 0
    - » int WW: # waiting writers; initially = 0
    - » Condition okToRead = NIL
    - » Condition okToWrite = NIL

```
Reader() {
  DataBase.BeginRead()
  // Now we are active!
  // Perform actual
  //read-only access
  AccessDatabase(ReadOnly);
  DataBase.EndRead();
}
Writer() {
  DataBase.BeginWrite()
  // Now we are active!
  // Perform actual
  //read/write access
  AccessDatabase(ReadWrite);
  DataBase.EndWrite();
}
```

Review : DataBase Monitor’s operations

```
BeginRead() {
   while ((AW + WW) > 0) {          // -Is it safe to read?
      WR++;                          //   -No. Writers exist
      okToRead.wait();               //     ->Sleep on cond var
      WR--; // No longer waiting
   }
   AR++; // Now we are active!
 }
```

```
EndRead(){
   AR--;     // No longer active
   if (AR == 0 && WW > 0)           // No other active readers
      okToWrite.signal();            // Wake up one writer
}
```

```
  BeginWrite() {
     while ((AW + AR) > 0) {        // -Is it safe to write?
        WW++;                        //    -No. Active users exist
        okToWrite.wait();            //      -> Sleep on cond var
        WW--; // No longer waiting
     }
     AW++;    // Now we are active!
  }
```

```
 EndWrite() {
     AW--;    // No longer active
     if (WW > 0){                    // Give priority to writers
        okToWrite.signal();          // Wake up one writer
     } else if (WR > 0) {            // Otherwise, wake reader
        okToRead.broadcast();        // Wake all readers
     }
  }
```

Simulation of Readers/Writers solution

- Consider the following sequence of operators:
  - R1, R2, W1, R3
- On entry, each reader checks the following:

```
while ((AW + WW) > 0) {  // Is it safe to read?
   WR++;                 // No. Writers exist
   okToRead.wait();      // Sleep on cond var
   WR--;                 // No longer waiting
}
AR++;                    // Now we are active!
```

- First, R1 comes along:
  AR = 1, WR = 0, AW = 0, WW = 0
- Next, R2 comes along:
  AR = 2, WR = 0, AW = 0, WW = 0
- Now, readers may take a while to access database
  - Situation: Locks released
  - Only AR is non-zero

Simulation(2)

- **Next, W1 comes along:**

```
while ((AW + AR) > 0) { // Is it safe to write?
   WW++;                // No. Active users exist
   okToWrite.wait();    // Sleep on cond var
   WW--;                // No longer waiting
}
AW++;
```

- **Can't start because of readers, so go to sleep:**

  AR = 2, WR = 0, AW = 0, WW = 1

- **Finally, R3 comes along:**
  AR = 2, WR = 1, AW = 0, WW = 1

- **Now, say that R2 finishes before R1:**
  AR = 1, WR = 1, AW = 0, WW = 1

- **Finally, last of first two readers (R1) finishes and wakes up writer:**

```
if (AR == 0 && WW > 0)  // No other active readers
   okToWrite.signal();  // Wake up one writer
```

Simulation(3)

- When writer wakes up, get:
  AR = 0, WR = 1, AW = 1, WW = 0
- Then, when writer finishes:

```
if (WW > 0){            // Give priority to writers
   okToWrite.signal();// Wake up one writer
} else if (WR > 0) { // Otherwise, wake reader
   okToRead.broadcast();    // Wake all readers
}
```

  - Writer wakes up reader, so get:
    AR = 1, WR = 0, AW = 0, WW = 0
- When reader completes, we are finished

Questions

- **Can readers starve? Consider BeginRead() code:**

```
while ((AW + WW) > 0) { // Is it safe to read?
   WR++;                // No. Writers exist
   okToRead.wait();     // Sleep on cond var
   WR--;                // No longer waiting
}
AR++;                   // Now we are active!
```

- **What if we erase the condition check in EndRead()?**

```
AR--;                   // No longer active
if (AR == 0 && WW > 0)  // No other active readers
   okToWrite.signal();  // Wake up one writer
```

- **Further, what if we turn the signal() into broadcast()**

```
AR--;                   // No longer active
okToWrite.broadcast();  // Wake up all writers
```

- **Finally, what if we use only one condition variable (call it "`okToContinue`") instead of two separate ones?**
  - **Both readers and writers sleep on this variable**
  - **Must use broadcast() instead of signal()**

Can we construct Monitors from Semaphores?

- **Locking aspect is easy: Just use a mutex**
- **Can we implement condition variables this way?**

```
Wait()   { semaphore.P(); }
Signal() { semaphore.V(); }
```

- **Does this work better?**

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
```

Construction of Monitors from Semaphores (con't)

- **Problem with previous try:**
  - P and V are commutative – result is the same no matter what order they occur
  - Wait and Signal on condition variables are NOT commutative
- **Does this fix the problem?**

```
Wait(Lock lock) {
   lock.Release();
   semaphore.P();
   lock.Acquire();
}
Signal() {
   if semaphore queue is not empty
      semaphore.V();
}
```

  - Not legal to look at contents of semaphore queue
  - There is a race condition – signaler can slip in after lock release and before waiter executes semaphore.P()
- **It is actually possible to do this correctly**
  - Complex solution for Hoare scheduling in book
  - Can you come up with simpler Mesa-scheduled solution?

Monitor Conclusion

- Monitors represent the logic of the program
  - Wait if necessary
  - Signal when change something so any waiting threads can proceed

- Basic structure of monitor-based program:

```
                          Check and/or update
Use monitor procedure  }    state variables
                           Wait if necessary

Do something so no need to wait


                          Check and/or update
Use monitor procedure  }    state variables
```

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:

```
class Account {
  private int balance;
  // object constructor
  public Account (int initialBalance) {
    balance = initialBalance;
  }
  public synchronized int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```

  - Every object has an associated lock which gets automatically acquired and released on entry and exit from a *synchronized* method.

Java Language Support for Synchronization (con’t)

- **Java also has *synchronized* statements:**

```
synchronized (object) {
        …
}
```

  - **Since every Java object has an associated lock, this type of statement acquires and releases the object’s lock on entry and exit of the body**
  - **Works properly even with exceptions:**

```
synchronized (object) {
    …
    DoFoo();
    …
}
void DoFoo() {
    throw errException;
}
```

Java Language Support for Synchronization (con't 2)

- In addition to a lock, every object has a single condition variable associated with it
  - How to wait inside a synchronization method or block:
    - `void wait(long timeout); // Wait for timeout`
    - `void wait(long timeout, int nanoseconds); //variant`
    - `void wait();`
  - How to signal in a synchronized method or block:
    - `void notify();    // wakes up oldest waiter`
    - `void notifyAll(); // like broadcast, wakes everyone`
  - Condition variables can wait for a bounded length of time. This is useful for handling exception cases:

    ```
    t1 = time.now();
    while (!ATMRequest()) {
       wait (CHECKPERIOD);
       t2 = time.new();
       if (t2 - t1 > LONG_TIME) checkMachine();
    }
    ```

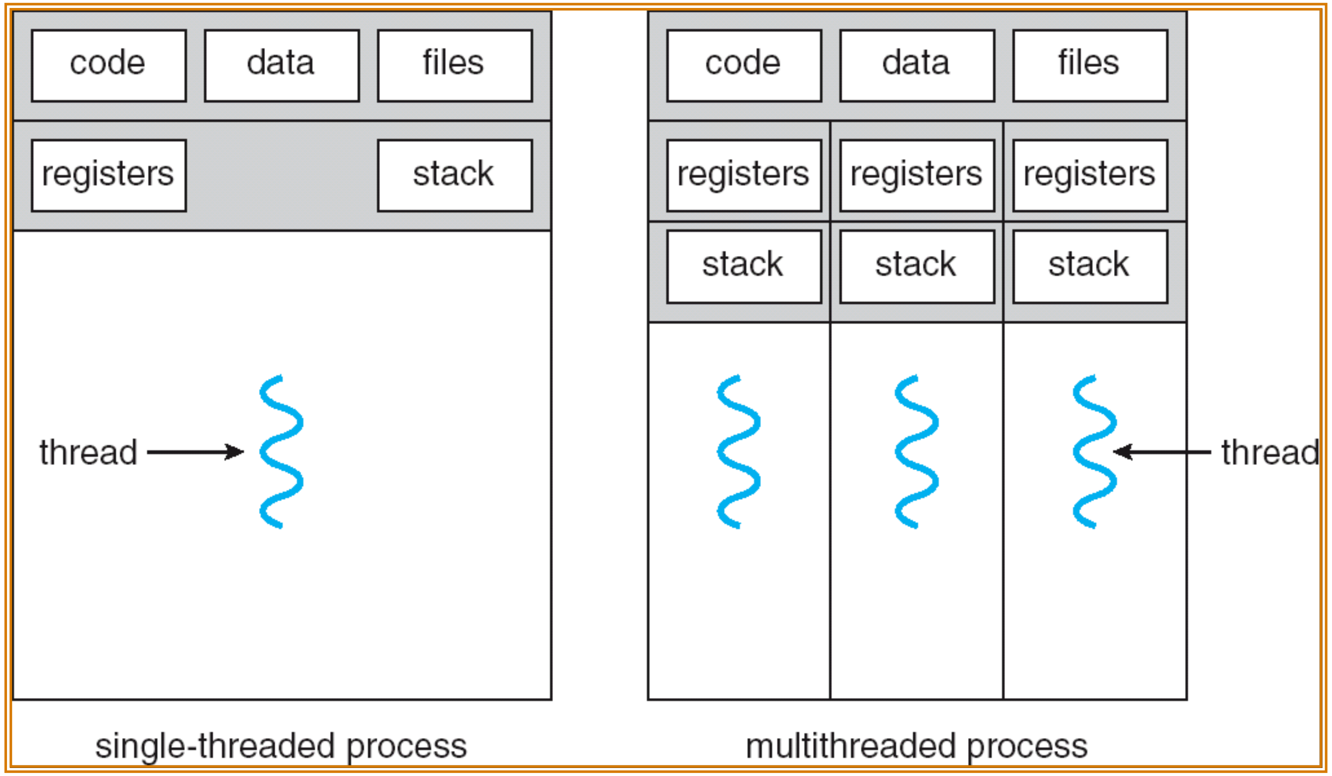
  - Not all Java VMs equivalent!
    - Different scheduling policies, not necessarily preemptive!

Memory Management

- Address binding
- Address translation
- Virtual memory

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)

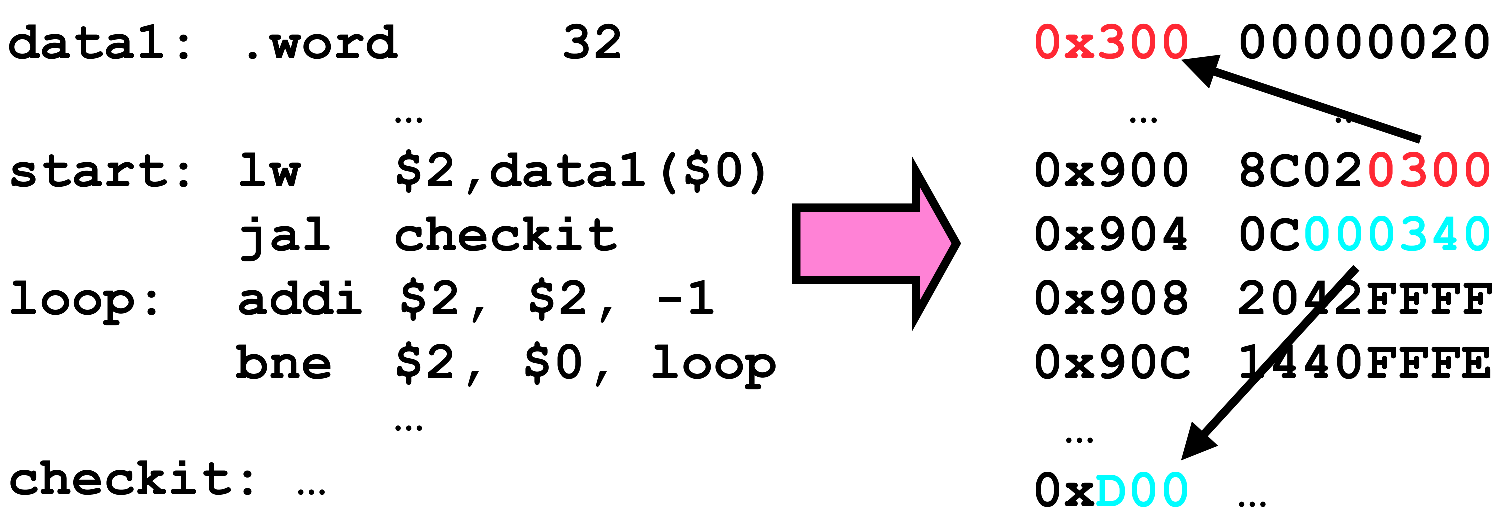
Recall: Single and Multithreaded Processes



- **Threads encapsulate concurrency**
  - **"Active" component of a process**
- **Address spaces encapsulate protection**
  - **Keeps buggy program from trashing the system**
  - **"Passive" component of a process**

Binding of Instructions and Data to Memory

- **Binding of instructions and data to addresses:**
  - Choose addresses for instructions and data from the standpoint of the processor



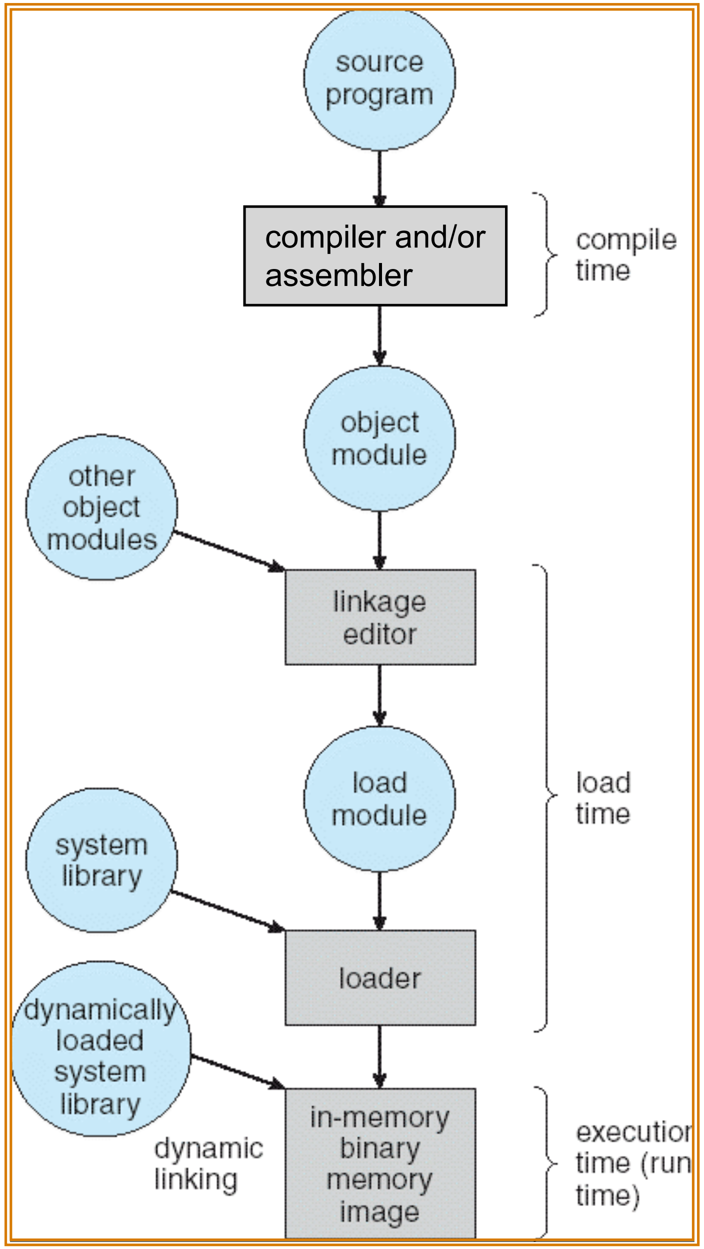
```
data1:  .word     32
              …
start:  lw    $2,data1($0)
        jal   checkit
loop:   addi  $2, $2, -1
        bne   $2, $0, loop
              …
checkit: …
```

```
0x300  00000020
  …      …
0x900  8C020300
0x904  0C000340
0x908  2042FFFF
0x90C  1440FFFE
 …
0xD00  …
```

  - Could we place `data1`, `start`, and/or `checkit` at different addresses?
    - » Yes. But need to modify some instructions or even data
      - ➢ Absolute addresses have to be relocated
    - » When?
      - Compile time/Load time/Execution time
  - Related: which physical memory locations hold particular instructions or data?

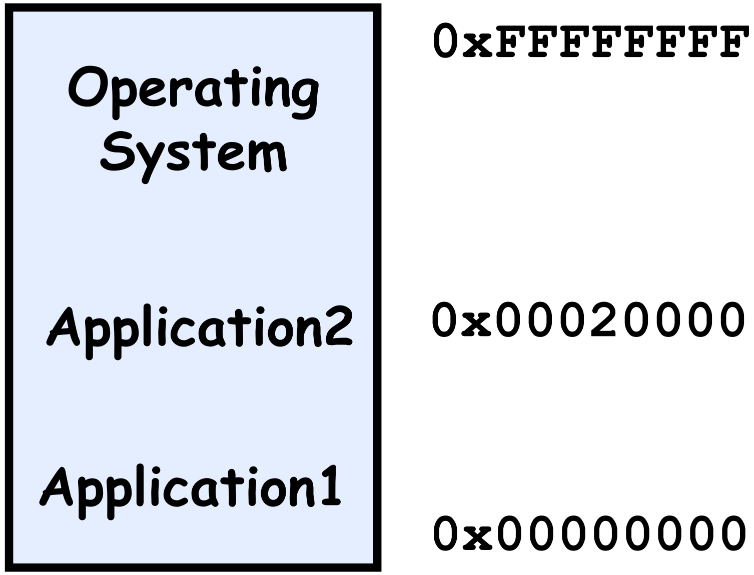
Multi-step Processing of a Program for Execution

- Preparation of a program for execution involves components at:
  - Compile and/or assembler time (i.e. “gcc” and or “as”)
  - Link/Load time (unix “ld” does link)
  - Execution time (e.g. dynamic libs)
- Addresses can be bound to final values anywhere in this path
  - Depends on hardware support
  - Also depends on operating system
- Dynamic Libraries
  - Linking postponed until execution
  - Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes routine



Multiprogramming (First Version)

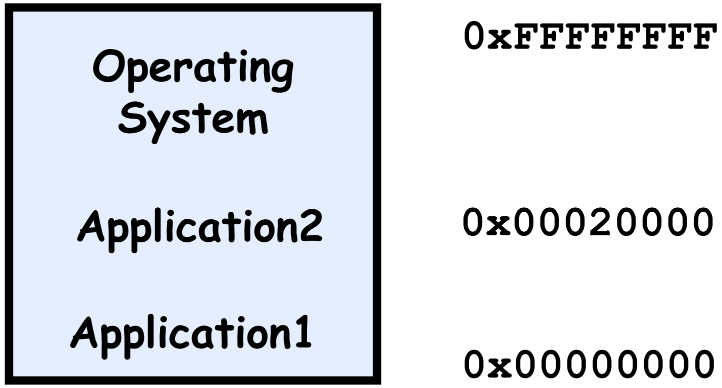
- **Multiprogramming without Translation or Protection**
  - Must somehow prevent address overlap between threads

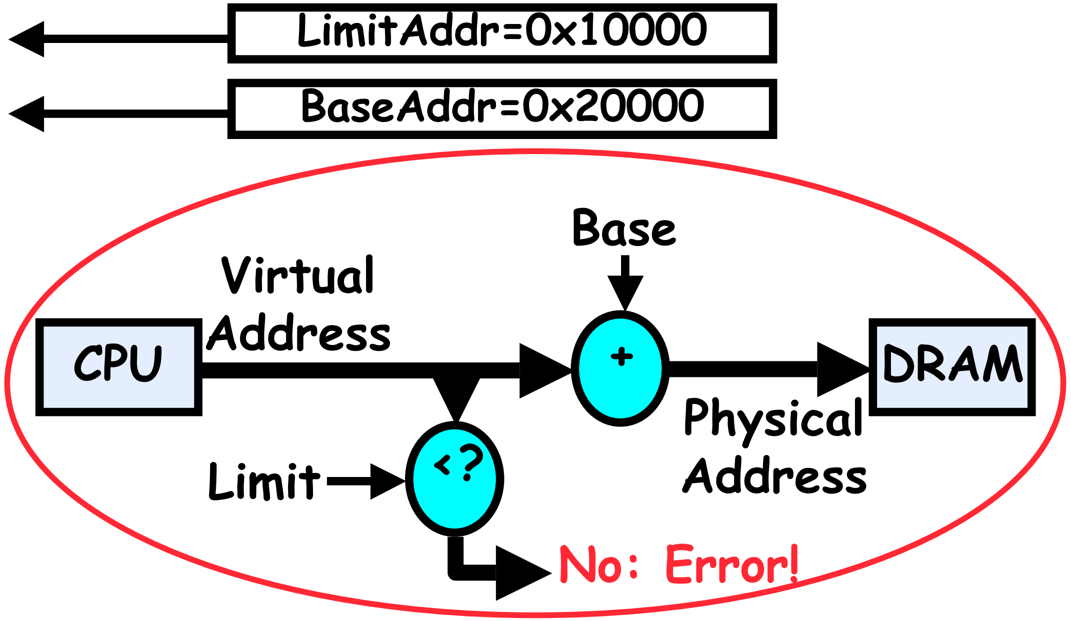


  - Trick: Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
    - » Everything adjusted to memory location of program
    - » Translation done by a linker-loader
    - » Was pretty common in early days
- **With this solution, no protection: bugs in any program can cause other programs to crash or even the OS**

Multiprogramming (Version with Protection)

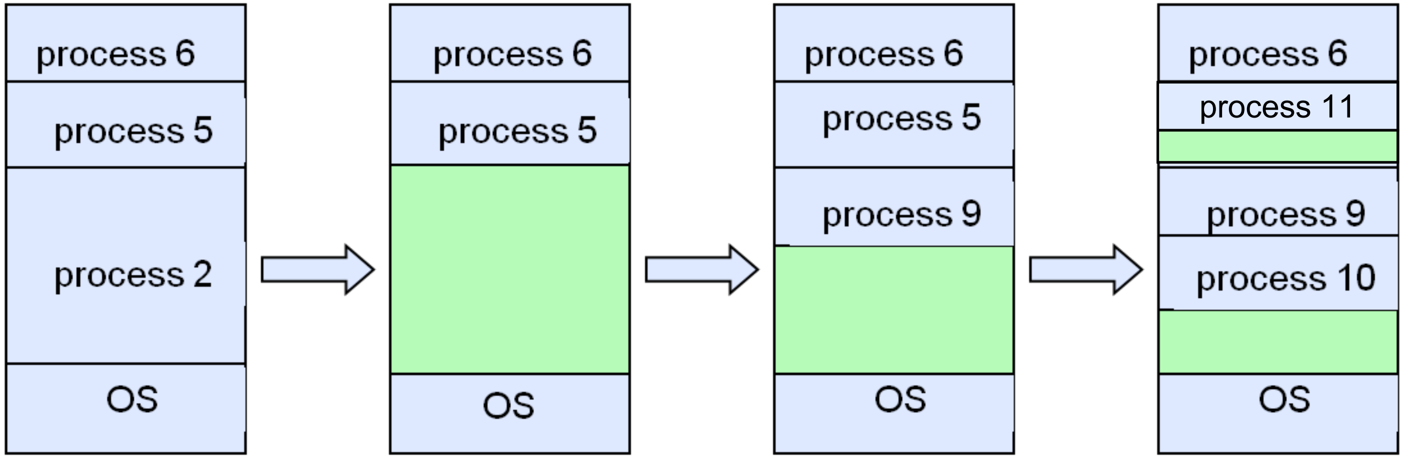
- Can we protect programs from each other without translation?





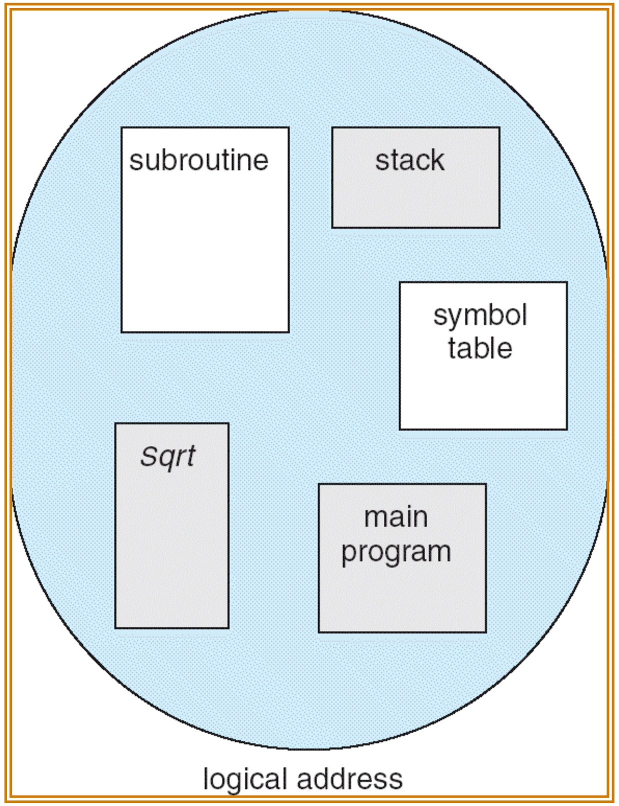
- ➢ Yes: use two special registers *BaseAddr* and *LimitAddr* to prevent user from straying outside designated area (segment)
  ==> Segmentation
  - – If user tries to access an illegal address, cause an error
  - – User may have multiple segments available (e.g x86)
    - » Loads and stores include segment ID in opcode:
      x86 Example: `mov [es:bx],ax.`
    - » Operating system moves around segment base pointers as necessary
  - – During switch, kernel loads new base/limit from TCB
    - » User not allowed to change base/limit registers

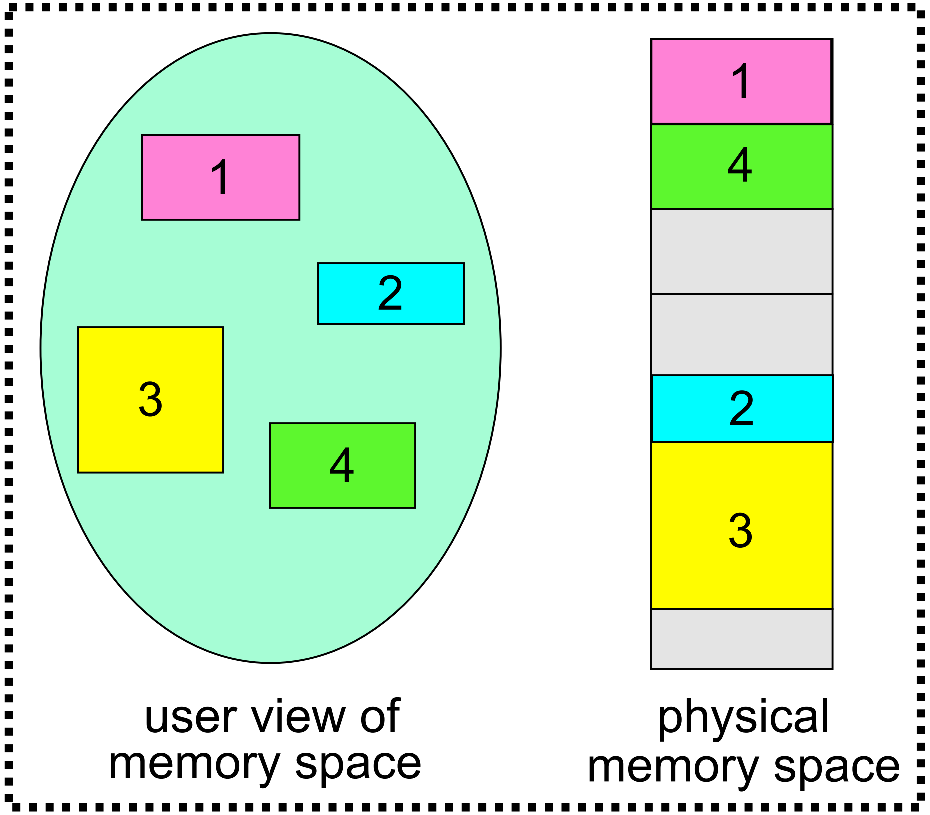
Issues with simple segmentation method



- **Fragmentation problem**
  - **Not every process is the same size**
  - **Over time, memory space becomes fragmented**
- **Need enough physical memory for every process**
  - **Doesn’t allow heap and stack to grow independently**
  - **Want to put these as far apart in memory as possible so that they can grow as needed**
- **Hard to do inter-process sharing**
  - **Want to share code segments when possible**
  - **Want to share memory between processes**
  - **Helped by by providing multiple segments per process**

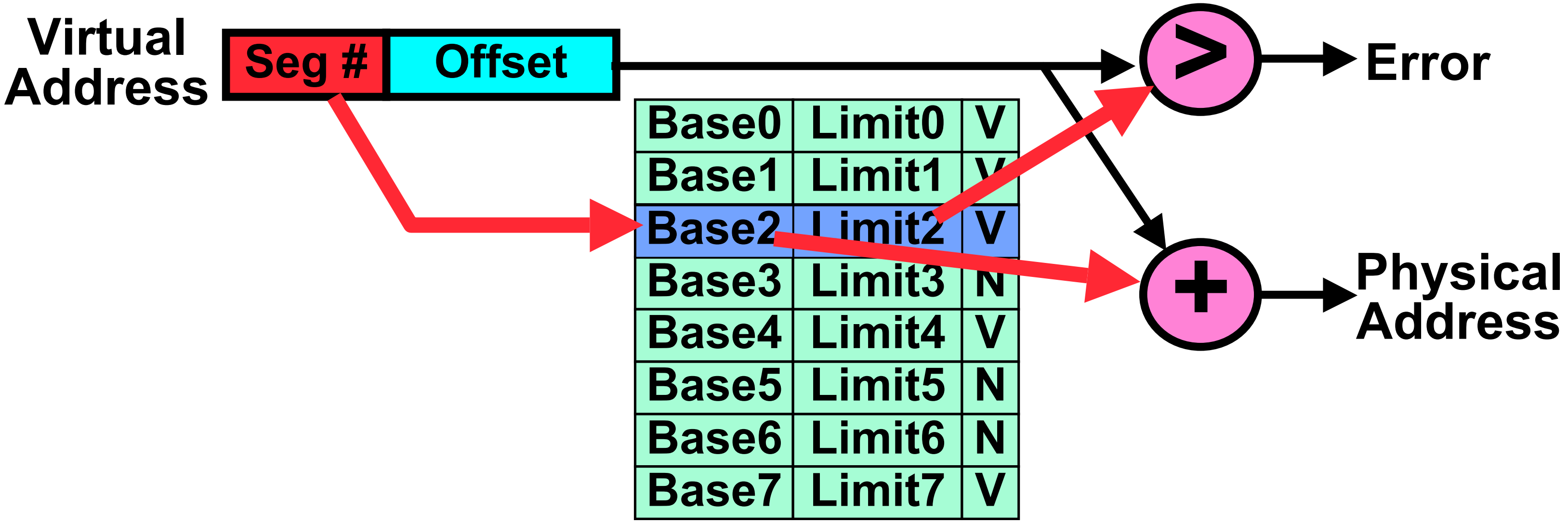
More Flexible Segmentation





- **Logical View: multiple separate segments**
  - **Typical: Code, Data, Stack**
  - **Others: memory sharing, etc**
- **Each segment is given region of contiguous memory**
  - **Has a base and limit**
  - **Can reside anywhere in physical memory**

Implementation of Multi-Segment Model



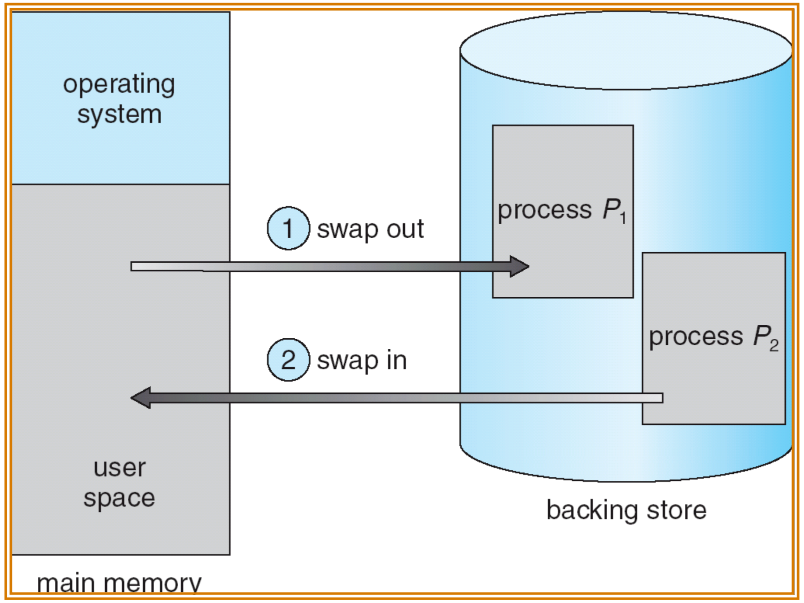
Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    - x86 Example: `mov [es:bx],ax.`
- What is “V/N”?
  - Can mark segments as invalid; requires check as well

Observations about Segmentation

- Virtual address space has holes
  - Segmentation efficient for sparse address spaces
  - A correct program should never address gaps (except as mentioned in moment)
    - If it does, trap to kernel and dump core
- When it is OK to address outside valid range:
  - This is how the stack and heap are allowed to grow
  - For instance, stack takes fault, system automatically increases size of stack
- Need protection mode in segment table
  - For example, code segment would be read-only
  - Data and stack would be read-write (stores allowed)
  - Shared segment could be read-only or read-write
- What must be saved/restored on context switch?
  - Segment table stored in CPU, not in memory (small)
  - Might store all of processes memory onto disk when switched (called "swapping")

Schematic View of Swapping

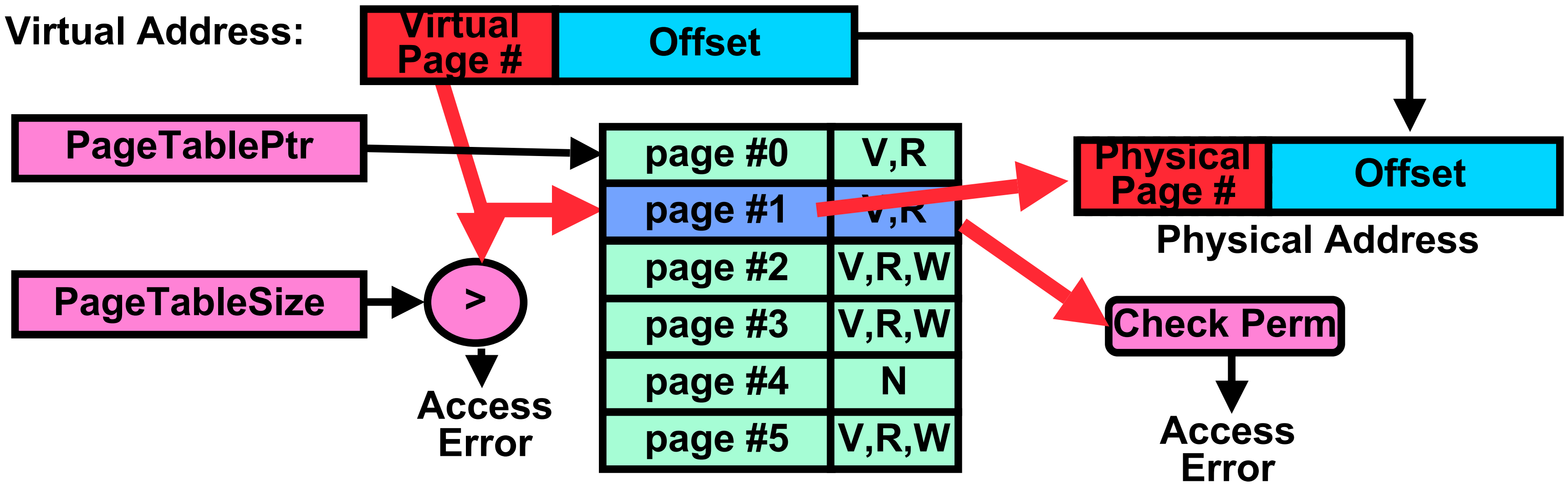


- **Extreme form of Context Switch: Swapping**
  - **In order to make room for next process, some or all of the previous process is moved to disk**
    - » **Likely need to send out complete segments**
  - **This greatly increases the cost of context-switching**
- **Desirable alternative?**
  - **Some way to keep only active portions of a process in memory at any one time**
  - **Need finer granularity control over physical memory**

Paging: Physical Memory in Fixed Size Chunks

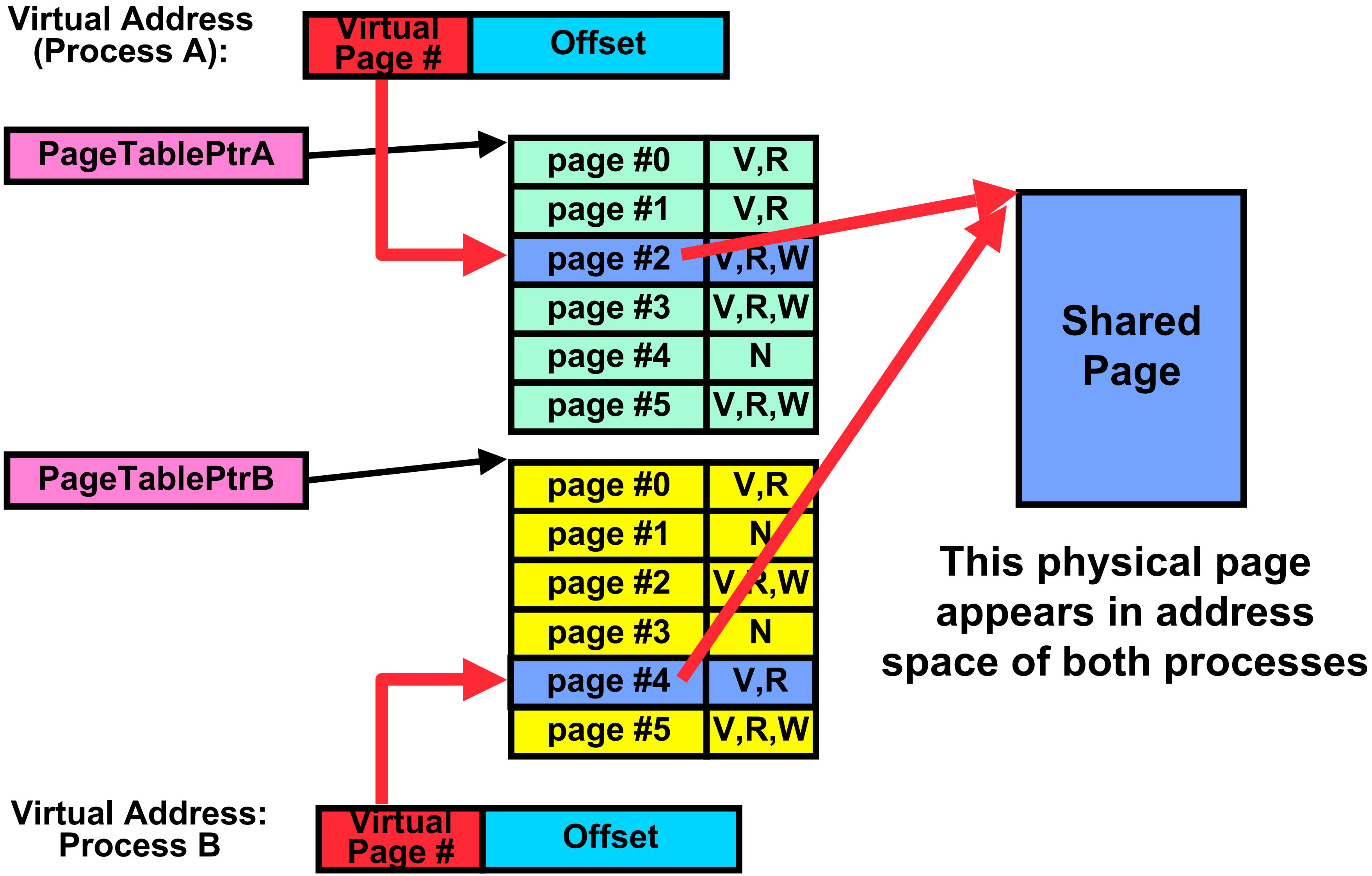
- Problems with segmentation?
  - Must fit variable-sized chunks into physical memory
  - May move processes multiple times to fit everything
  - Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don't need all memory within allocated chunks
- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    - » Can use simple vector of bits to handle allocation:
      0011000111000110 … 110010
    - » Each bit represents page of physical memory
      1⇒allocated, 0⇒free
- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    - » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment

How to Implement Paging?



- Page Table (One per process)
  - Residues in physical memory
  - Contains physical page and permission for each virtual page
    - Permissions include: Valid bits, Read, Write, etc
- Virtual address mapping
  - Offset from Virtual address copied to Physical Address
    - Example: 10 bit offset ⇒ 1024-byte pages
  - Virtual page # is all remaining bits
    - Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
    - Physical page # copied from table into physical address
  - Check Page Table bounds and permissions

What about Sharing?

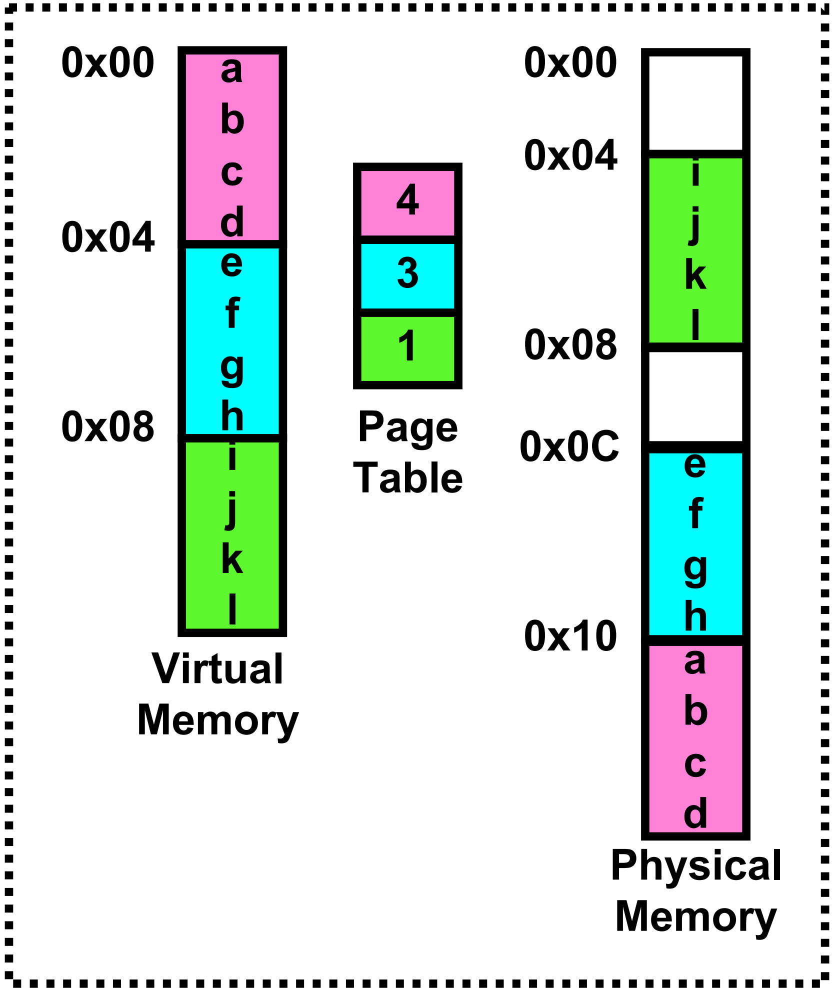


page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

page #0	V,R
page #1	N
page #2	V,R,W
page #3	N
page #4	V,R
page #5	V,R,W

This physical page appears in address space of both processes

Simple Page Table Discussion

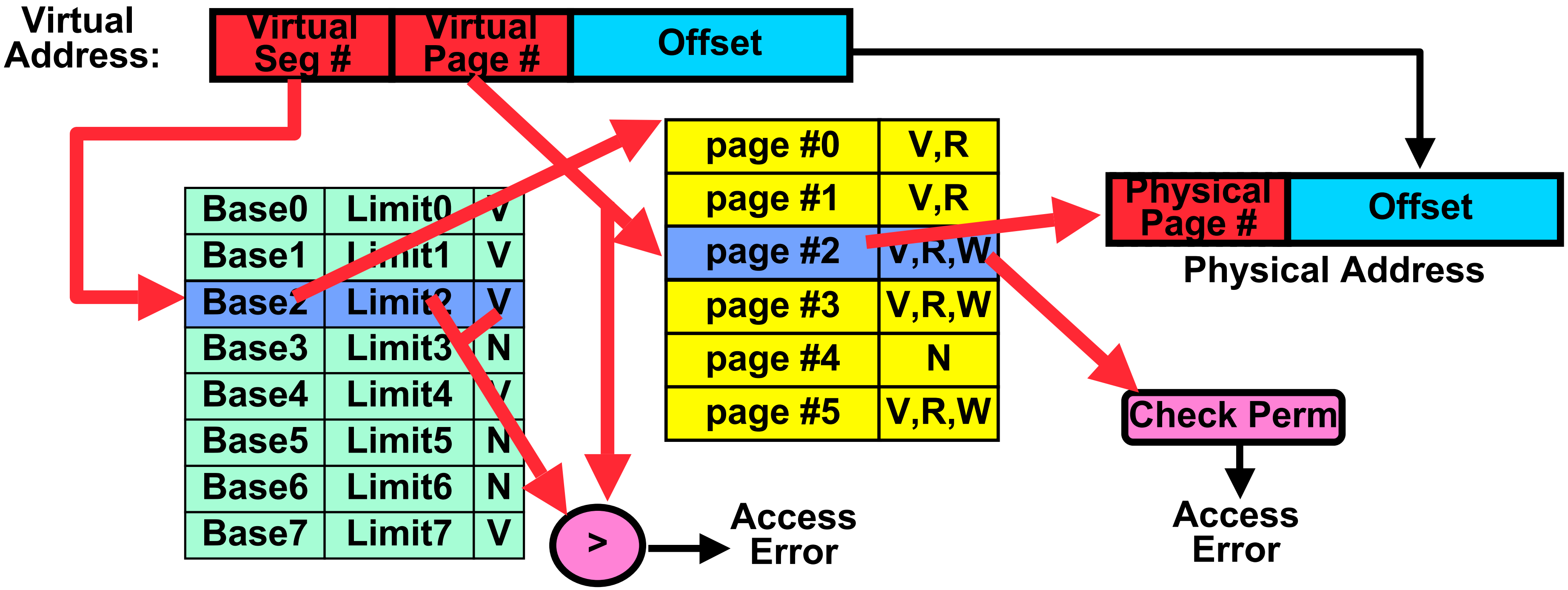


Example (4 byte pages)

- What needs to be switched on a context switch?
  - Page table pointer and limit
- Analysis
  - Pros
    - » Simple memory allocation
    - » Easy to Share
  - Con: What if address space is sparse?
    - » E.g. on UNIX, code starts at 0, stack starts at (2^{31}-1).
    - » With 1K pages, need 4 million page table entries!
  - Con: What if table really big?
    - » Not all pages used all the time ⇒ would be nice to have working set of page table in memory
- How about combining paging and segmentation?

Multi-level Translation

- **What about a tree of tables?**
  - **Lowest level page table⇒memory still allocated with bitmap**
  - **Higher levels often segmented**
- **Could have any number of levels. Example (top segment):**

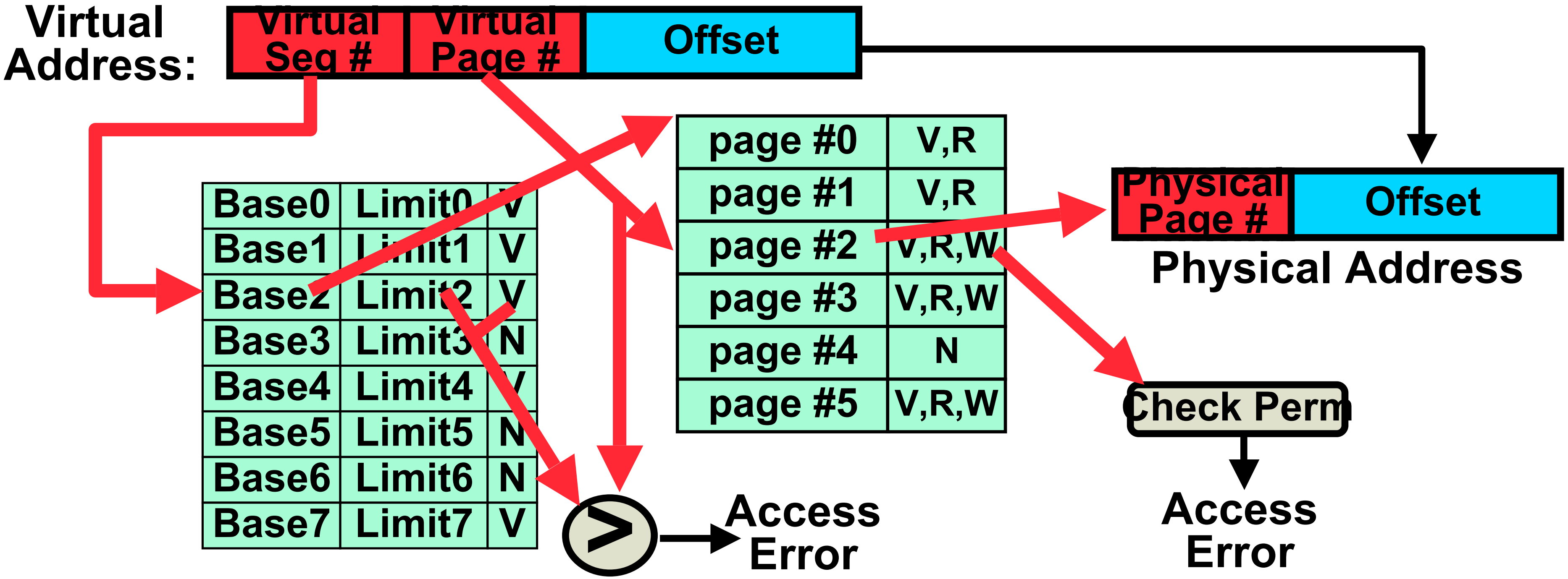


Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

- **What must be saved/restored on context switch?**
  - **Contents of top-level segment registers (for this example)**
  - **Pointer to top-level table (page table)**

How long does Address translation take ?



Virtual Address:

Virtual Seg #	Virtual Page #	Offset

Base0	Limit0	V
Base1	Limit1	V
Base2	Limit2	V
Base3	Limit3	N
Base4	Limit4	V
Base5	Limit5	N
Base6	Limit6	N
Base7	Limit7	V

page #0	V,R
page #1	V,R
page #2	V,R,W
page #3	V,R,W
page #4	N
page #5	V,R,W

Physical Page #	Offset

Physical Address

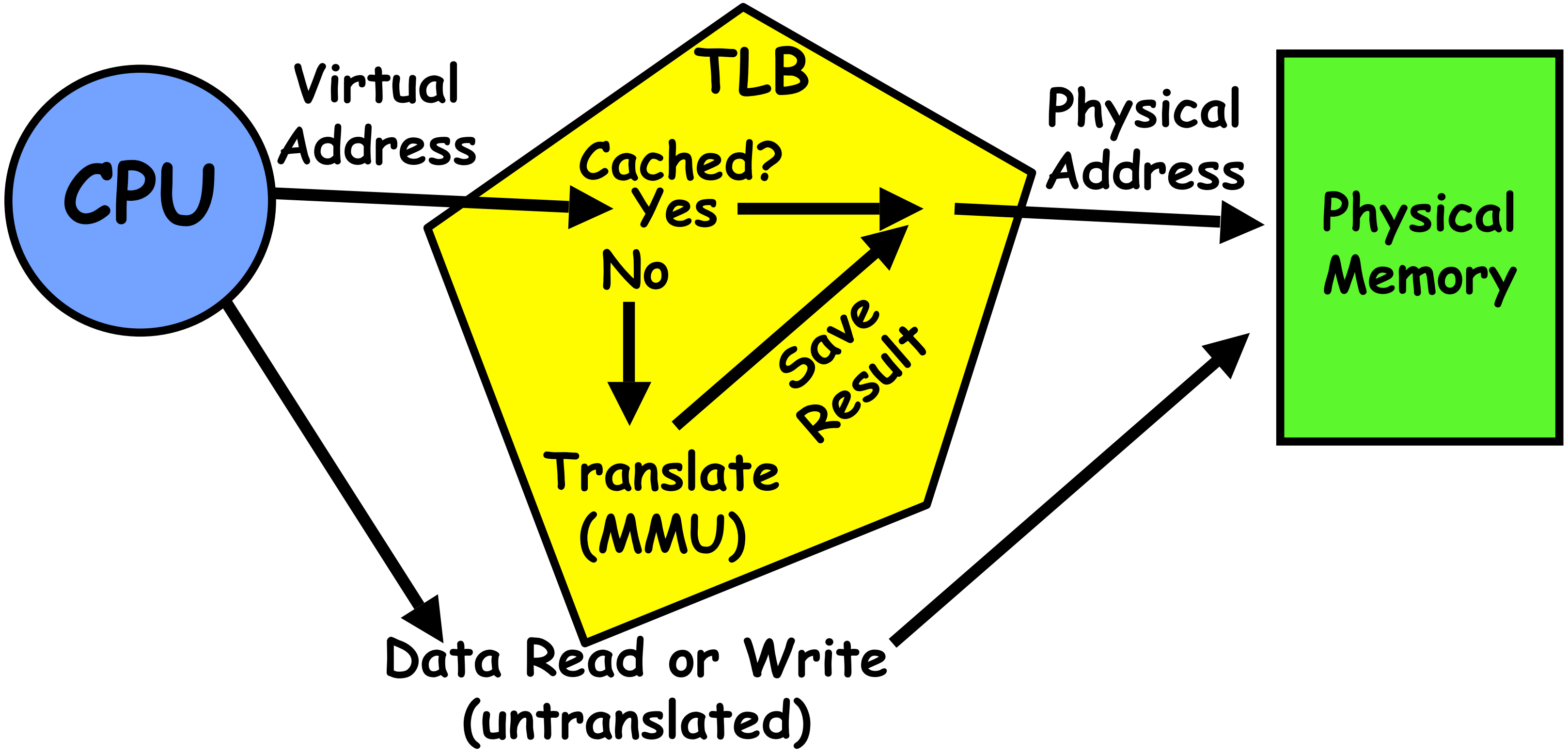
Access Error

Check Perm

Access Error

- Cannot afford to translate on every access
  - At least three DRAM accesses per actual DRAM access
  - Or: perhaps I/O if page table partially on disk!
- Even worse: What if we are using caching to make memory access faster than DRAM access???
- Solution? Cache translations!
  - Translation Cache: TLB (“Translation Lookaside Buffer”)

Caching Applied to Address Translation



- **Question is one of page locality: does it exist?**
  - **Instruction accesses spend a lot of time on the same page (since accesses sequential)**
  - **Stack accesses have definite locality of reference**
  - **Data accesses have less page locality, but still some…**

Summary

- Memory is a resource that must be shared
  - Controlled Overlap: only shared when appropriate
  - Translation: Change Virtual Addresses into Physical Addresses
  - Protection: Prevent unauthorized Sharing of resources
- Simple Protection through Segmentation
  - Base+limit registers restrict memory accessible to user
  - Can be used to translate as well
- Full translation of addresses through Memory Management Unit (MMU)
  - Paging : Memory divided into fixed-sized chunks (pages) of memory
  - Virtual page number from virtual address mapped through page table to physical page number
  - Offset of virtual address same as physical address
  - Changing of page tables only available to kernel
  - Every Access translated through page table
    - » Translation speeded up using a TLB (cache for recent translations)
- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space