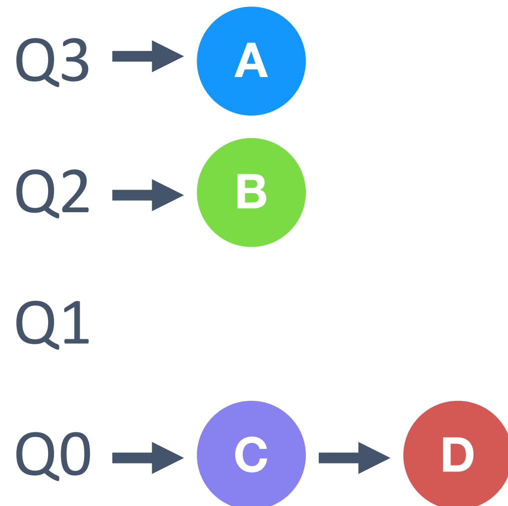
CPU Virtualization: Scheduling

Review of Concepts: Scheduling

Workloads:
arrival_time
run_time

Schedulers:
FIFO
SJF
STCF
RR

Metrics (efficiency):
turnaround_time
response_time

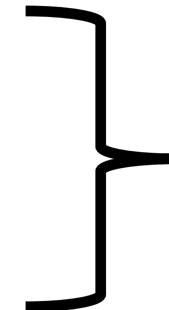


MLFQ

Fair sharing schedulers

Lottery

A: 100
B: 50
C: 250



Choose randomly in proportion to tickets

Stride Scheduling

- Can we make lottery scheduling more “deterministic”?
- Goal: Make each process run deterministically in proportion to its tickets
- Suppose tickets A: 100, B: 50, C: 250
- Define a stride: (large number, say 10000)/#tickets
  - A: 100, B: 200, C: 40
- Every time process runs, increment a counter, pass, by the stride
- Pick process with the minimum pass to schedule

Stride Scheduling example

Pass(A)	Pass(B)	Pass(C)	Who runs?
stride=100	200	40	
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A

Problems with stride scheduling

- Simple enough, but…
- Global state can be bad (across processes)
- Troublesome to provide fair share for new processes
- Suppose process D enters after 5 runs of stride scheduler
  - What should its pass value be?
  - Lottery: every scheduler run is likely to choose process proportional to its fair share
- If the process needs to change its priority (tickets), how to reinterpret its pass value?

Completely Fair Scheduling

- Goal: scheduling many 1000s of processes efficiently

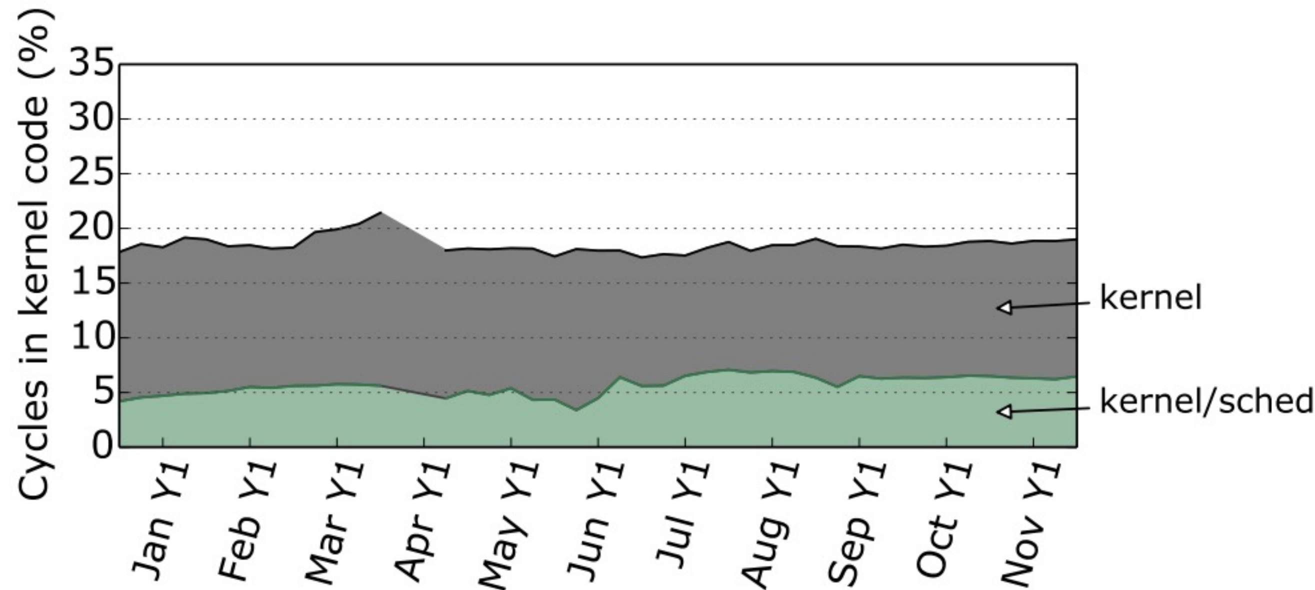


Figure 5: Kernel time, especially time spent in the scheduler, is a significant fraction of WSC cycles.

Profiling a warehouse-scale computer

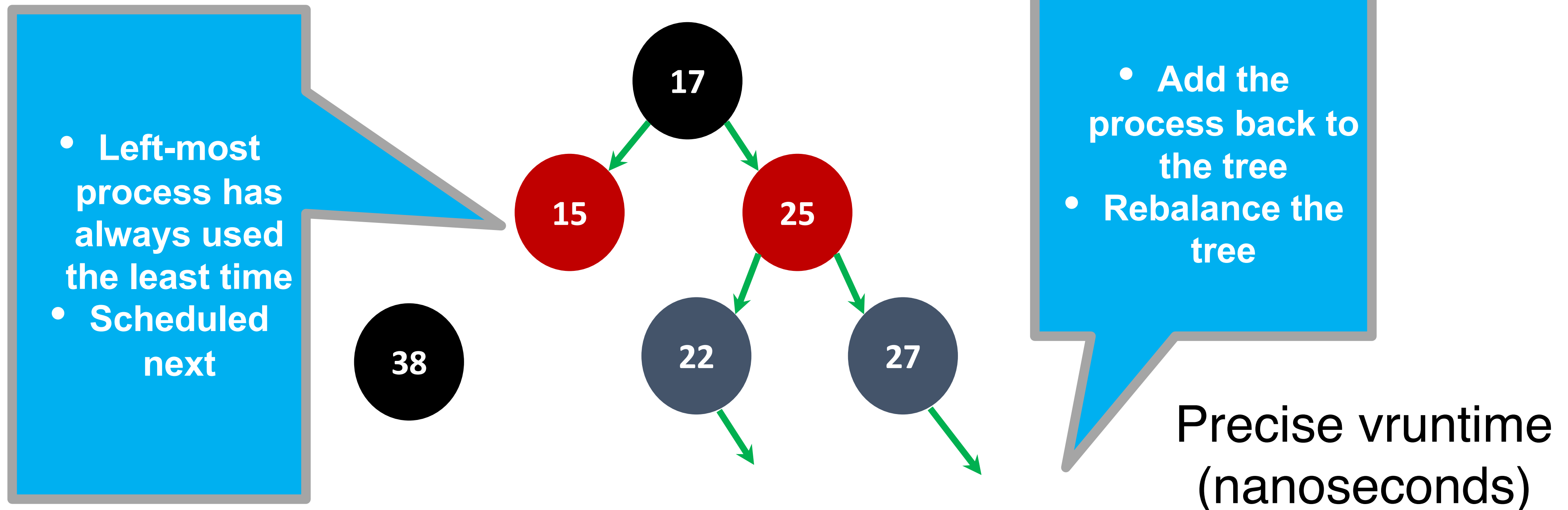
Svilen Kanev[†] Harvard University
Juan Pablo Darago[†] Universidad de Buenos Aires
Kim Hazelwood[†] Yahoo Labs
Parthasarathy Ranganathan Google
Tipp Moseley Google
Gu-Yeon Wei Harvard University
David Brooks Harvard University

Completely Fair Scheduler

- On Linux, in use since 2.6.23, has O(log N) runtime
- Move from MLFQ to Weighted Fair Queuing
  - First major OS to use a fair scheduling algorithm
  - Processes ordered by the amount of CPU time they use
- Gets rid of queues and linked lists in favor of a red-black tree of processes

Key ideas of CFS

- Maintain a counter of cumulative execution time
  - Virtual runtime, like the "pass" in stride scheduling
- Schedule process with least virtual runtime. Use R-B trees



Key ideas of CFS

- Don't use a fixed time slice per run
- Instead of stride, divide up a "time over which scheduler should be fair" into slices according to number of runnable processes
  - sched_latency
  - Like the "large number" we used to compute the stride previously
- But what if there are too many processes?
  - Spend more time context switching than executing processes
  - min_granularity
- Even if process time slice is not a multiple of the timer, can track vruntime precisely using the actual execution time

Nice levels to set priorities

```
ps ax -eo pid,ni,rtprio,cmd
```

- Nice value between -20 to 20. Equivalent of tickets

```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

$$\texttt{time_slice}_k = \frac{\texttt{weight}_k}{\sum_{i=0}^{n-1} \texttt{weight}_i} \cdot \texttt{sched_latency}$$

$$\texttt{vruntime}_i = \texttt{vruntime}_i + \frac{\texttt{weight}_0}{\texttt{weight}_i} \cdot \texttt{runtime}_i$$

Multi-Core Scheduling

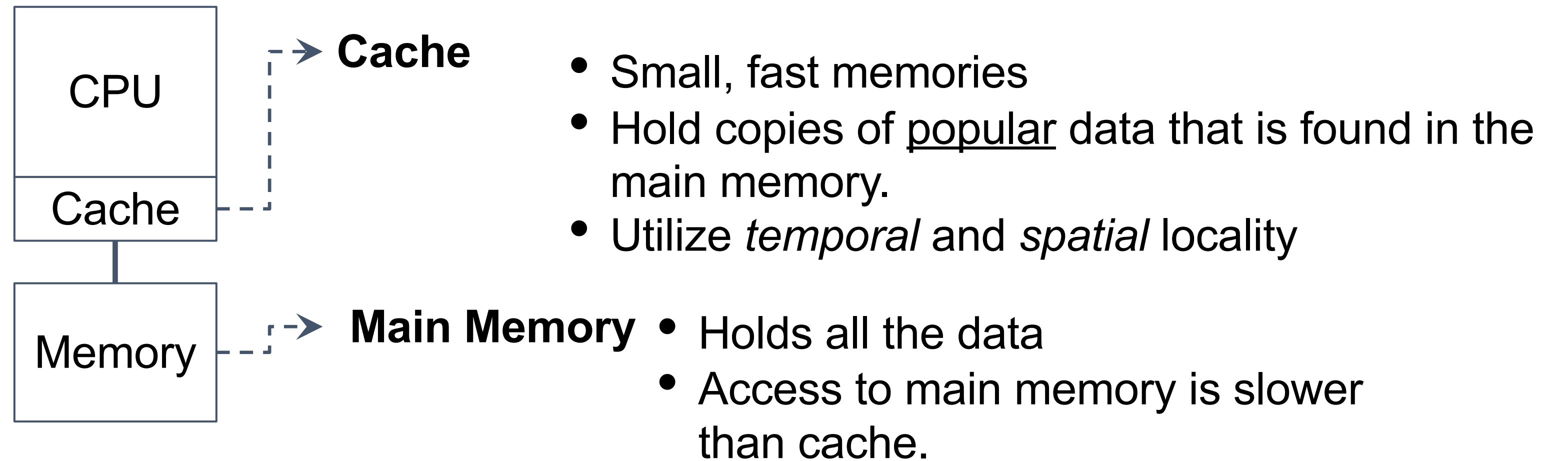
Multiprocessor Scheduling

- The rise of the multicore processor is the source of multiprocessor-scheduling proliferation.
  - **Multicore**: Multiple CPU cores are packed onto a single chip.

- Adding more CPUs does not make that single application run faster.
- Rewrite application to run in parallel, using **threads**.

How to schedule jobs on Multiple CPUs?

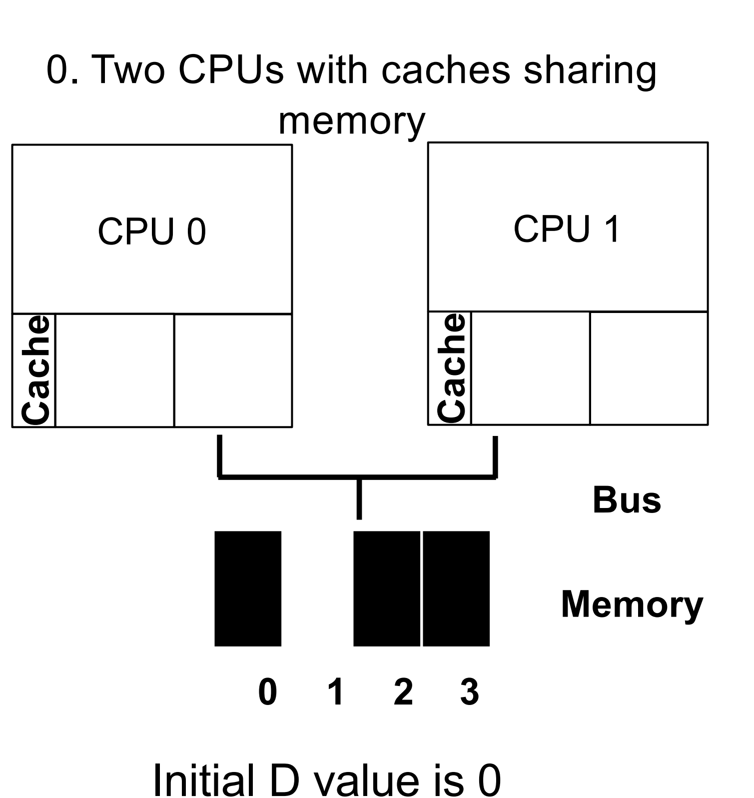
Single CPU with cache



By keeping data in cache, the system can make slow memory appear to be a fast one

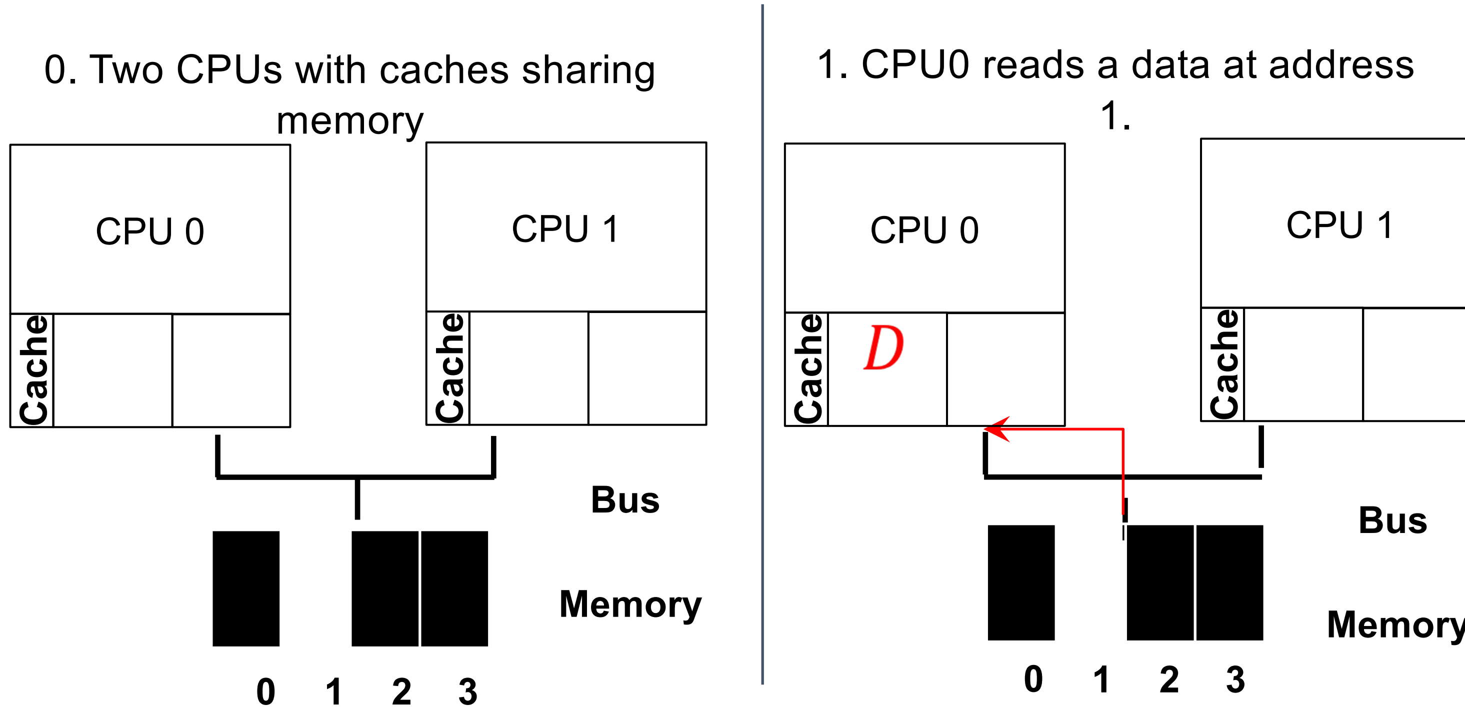
Cache Coherence

- Consistency of shared resource data stored in multiple caches.



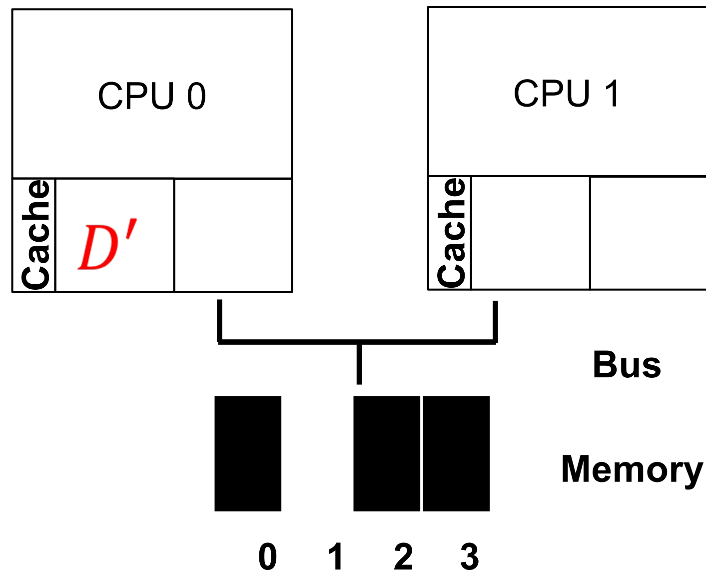
Cache Coherence

- Consistency of shared resource data stored in multiple caches.



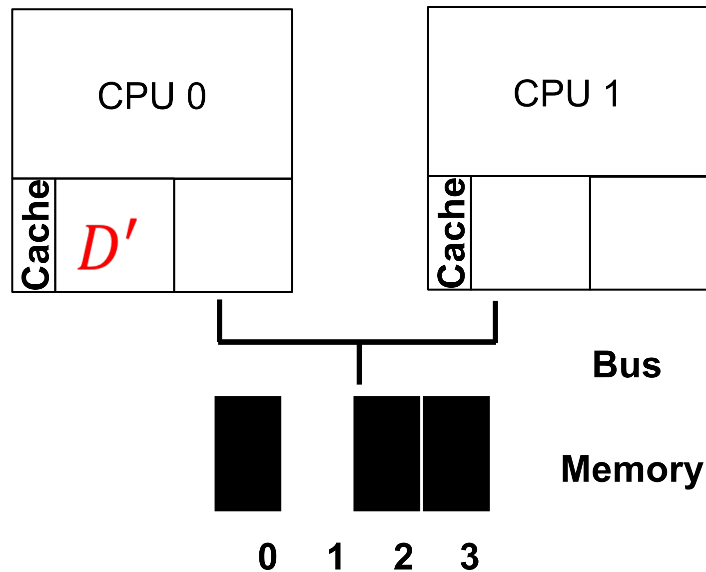
Cache Coherence

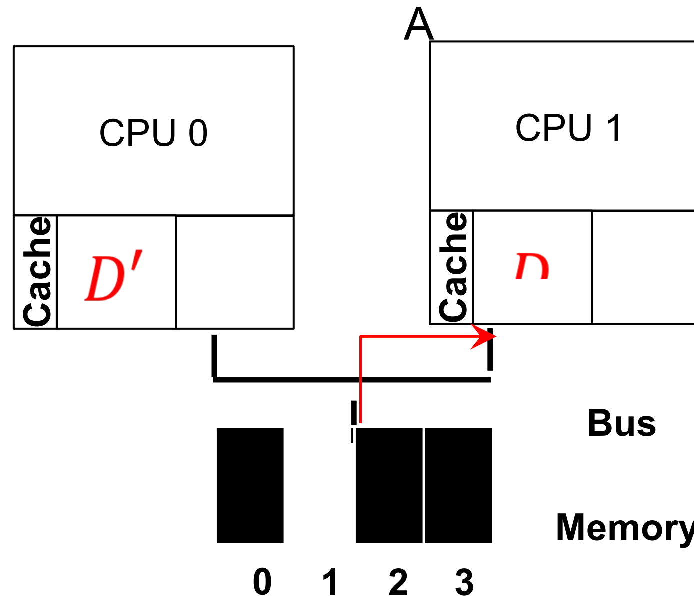
- Consistency of shared resource data stored in multiple caches.



Cache Coherence

- Consistency of shared resource data stored in multiple caches.





Data Inconsistency Problem!

Cache Coherence: one solution

- Bus snooping
  - Each cache pays attention to memory updates by **observing the bus**.
  - When a CPU sees an update for a data item it holds in its cache, it will notice the change and either invalidate its copy or update it.
- When accessing shared data across CPUs, mutual exclusion primitives should be used to guarantee correctness

Don’t forget synchronization

```
1       typedef struct __Node_t {
2               int value;
3               struct __Node_t *next;
4       } Node_t;
5
6       int List_Pop() {
7               Node_t *tmp = head;           // remember old head ...
8               int value = head->value;  // ... and its value
9               head = head->next;     // advance head to next pointer
10              free(tmp);             // free old head
11              return value;                 // return value at head
12      }
```

Don’t forget synchronization

```
1          typedef struct __Node_t {
2                  int value;
3                  struct __Node_t *next;
4          } Node_t;
5
6   int List_Pop() {
7          lock(&m)
8                  Node_t *tmp = head;           // remember old head ...
9                  int value = head->value;  // ... and its value
10                 head = head->next;     // advance head to next pointer
11             free(tmp);                     // free old head
11             unlock(&m)
12                 return value;                     // return value at head
13         }
```

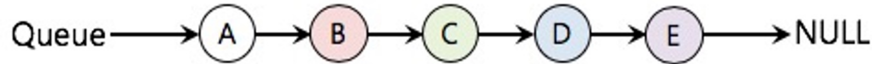
Cache Affinity

- Keep a process on the same CPU if at all possible
  - A process builds up a fair bit of state in the cache of a CPU.
  - The next time the process run, it will run faster if some of its state is *already present* in the cache on that CPU.

A multiprocessor scheduler should consider cache affinity when making its scheduling decision.

Cache Affinity

- Put all jobs that need to be scheduled into a single queue

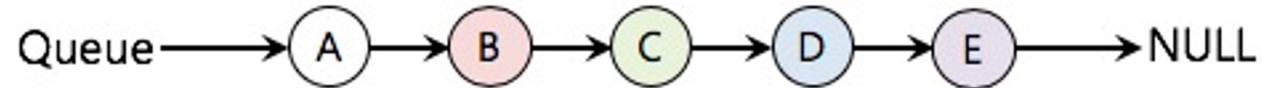


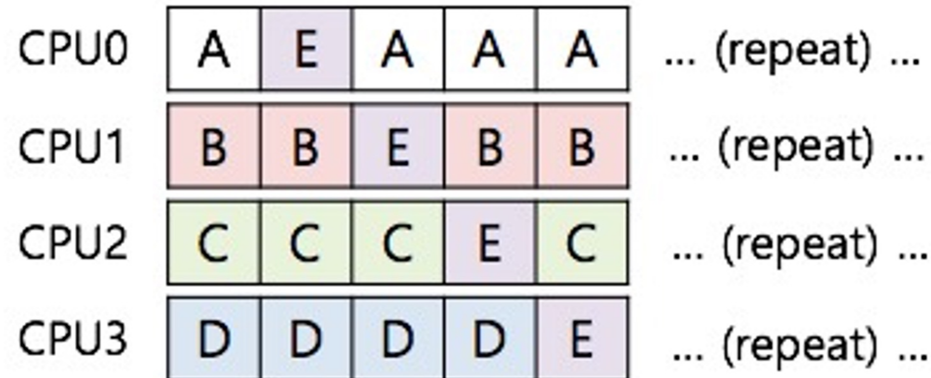
- Each CPU simply picks the next job from the globally shared queue. Simple.
- Cons:
  - Some form of **locking** has to be inserted
  - Lack of scalability
  - Cache affinity

Cache Affinity?

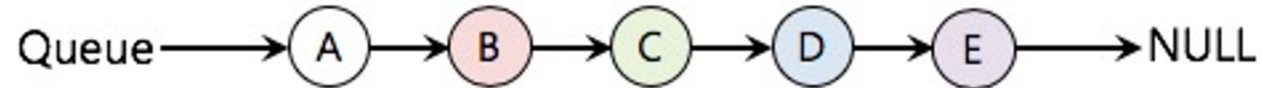
- No cache affinity ☹

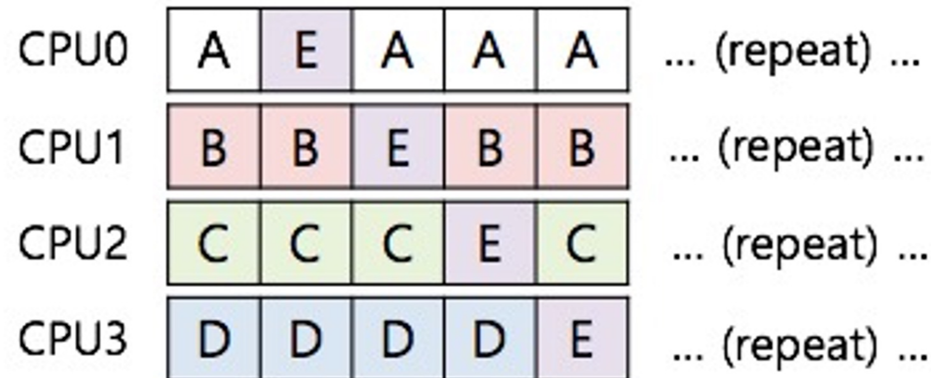
Scheduling with cache affinity





Scheduling with cache affinity





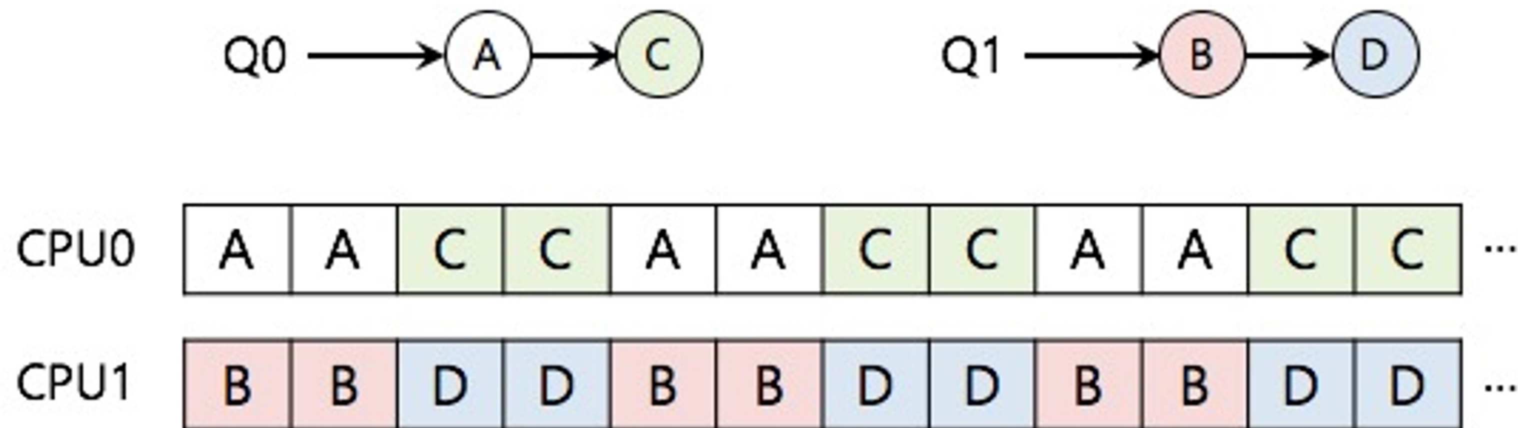
- Preserving affinity for most
  - Jobs A through D are not moved across processors.
  - Only job **E** Migrating from CPU to CPU.
- Implementing such a scheme can be **complex**.

Multi-queue Multiprocessor Scheduling (MQMS)

- MQMS consists of multiple scheduling queues.
  - Each queue will follow a particular scheduling discipline.
  - When a job enters the system, it is placed on **exactly one** scheduling queue.
  - Avoid the problems of information sharing and synchronization.

Multi-queue Multiprocessor Scheduling (MQMS)

- With **round robin**, the system might produce a schedule that looks like this:



MQMS provides more scalability and cache affinity.

Problem with MQMS?

- Load Imbalance

Need ways to balance load across cores over time by migrating processes across cores

Summary

Understand goals (metrics) and workload, then design scheduler around that

General purpose schedulers need to support processes with different goals

Past behavior is good predictor of future behavior

Random algorithms (lottery scheduling) can be simple to implement and avoid corner cases.

Multiprocessor scheduling: incorporate cache affinity & contention

Memory Virtualization

Questions answered

- What is in the address space of a process (review)?
- What are the different ways that that OS can virtualize memory?
    - Time sharing, static relocation, dynamic relocation (base, base + bounds, segmentation)
- What hardware support is needed for dynamic relocation?

More Virtualization

1st part of course: Virtualization

Virtual CPU: *illusion* of **private CPU registers**

Virtual RAM: *illusion* of **private memory**

Memory Virtualization – Then (1974)

Albrecht (an astronomer) sits with pencil paper working on code in front of a HP bulky computer, with one processor and a breathtaking **16 kilobytes** of magnetic-core memory and **1 Processor**

Developed virtual memory system to handle files on tape that were larger than the available memory on his Hewlett Packard 2116 microcomputer

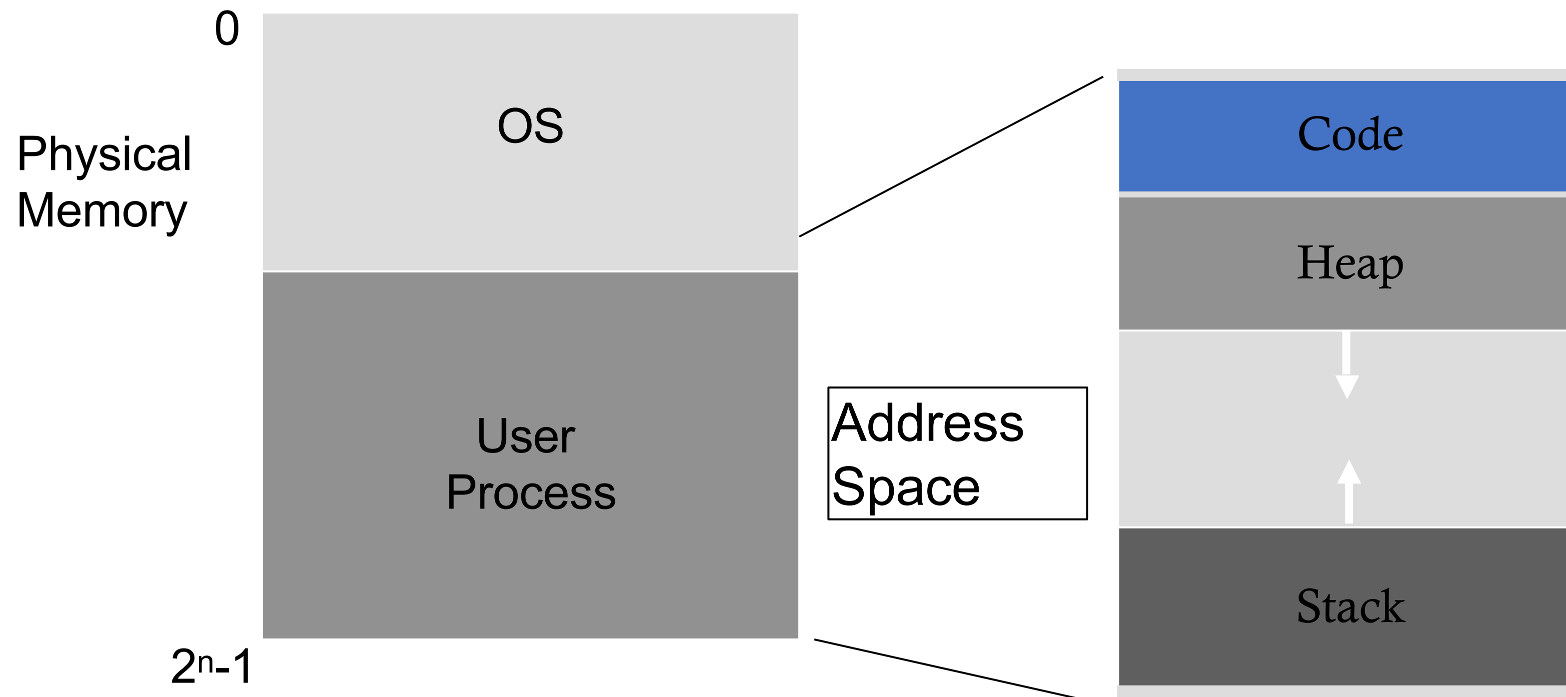
Memory Virtualization – recently (2013)

Albrecht (an astronomer, circa 2013) in front of a datacenter system with 40 processor cores, 138 terabytes of storage capacity and 83 gigabytes of RAM

5 million times more memory!

Motivation for Virtualization

Uniprogramming: One process runs at a time



Disadvantages:

- Only one process runs at a time
- Process can destroy OS

Goals of Mem Virtualization

Transparency

- Processes are not aware that memory is shared
- Works regardless of number and/or location of processes

Protection

- Cannot corrupt OS or other processes
- Privacy: Cannot read data of other processes

Efficiency

- Do not waste memory resources (minimize fragmentation)

Sharing

- Cooperating processes can share portions of address space

Abstraction:Address Space

Address space: Each process has set of addresses that map to bytes

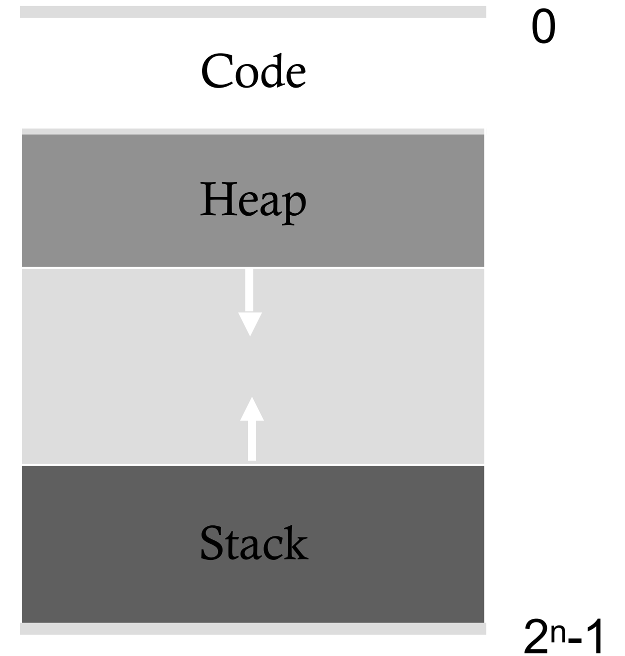
Problem:

How can OS provide illusion of private address space to each process?

Review:What is in an address space?

Address space has static and dynamic components

- Static: Code and some global variables
- Dynamic: Stack and Heap



Motivation for Dynamic Memory

Why do processes need dynamic allocation of memory?

- Do not know amount of memory needed at compile time
- Must be pessimistic when allocate memory statically
  - Allocate enough for worst possible case; Storage is used inefficiently

Recursive procedures

- Do not know how many times procedure will be nested

Complex data structures: lists and trees

- `struct my_t *p = (struct my_t *)malloc(sizeof(struct my_t));`

Two types of dynamic allocation

- Stack
- Heap

Where Are stacks Used?

OS uses stack for procedure call frames (local variables and parameters)

```
main () {
      int A = 0; foo (A);
      printf("A: %d\n", A);
}

void foo (int Z) {
      int A = 2; Z = 5;
      printf("A: %d Z: %d\n", A, Z);
}
```

Heap Organization

Definition: Allocate from any random location: malloc(), new()

- Heap memory consists of allocated areas and free areas (holes)
- Order of allocation and free is unpredictable

Advantage

- Works for all data structures

Disadvantages

- Allocation can be slow
- End up with small chunks of free space - fragmentation
- Where to allocate 12 bytes? 16 bytes? 24 bytes??

- What is OS's role in managing heap?
  - OS gives big chunk of free memory to process; library manages individual allocations

Quiz: Match that Address Location

```
int x;
int main(int argc, char *argv[]) {
  int y;
  int *z = malloc(sizeof(int)););
}
```

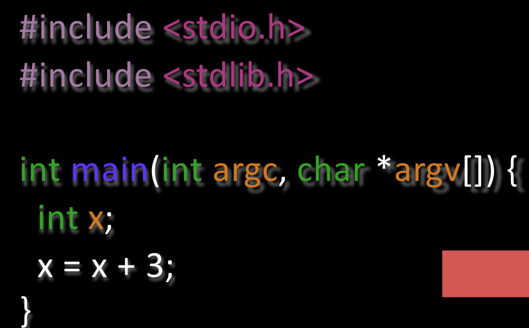
main? x?
y? z?
*z?

Possible locations: static data, code, stack, heap

What if no static data location?

Address	Location
x	Static data → Code
main	Code
y	Stack
z	Stack
*z	Heap

Memory Accesses




```
0x10:   movl0x8(%rbp), %edi
0x13:   addl $0x3, %edi
0x19:   movl%edi, 0x8(%rbp)
```

otool -tv demo1.o
(or objdump on Linux)

%**rbp** is the base pointer:
points to base of current stack frame

How to Virtualize Memory?

Problem: How to run multiple processes simultaneously?

Addresses are “hardcoded” into process binaries.

➔ How to avoid collisions?

Possible Solutions for Mechanisms (covered today):

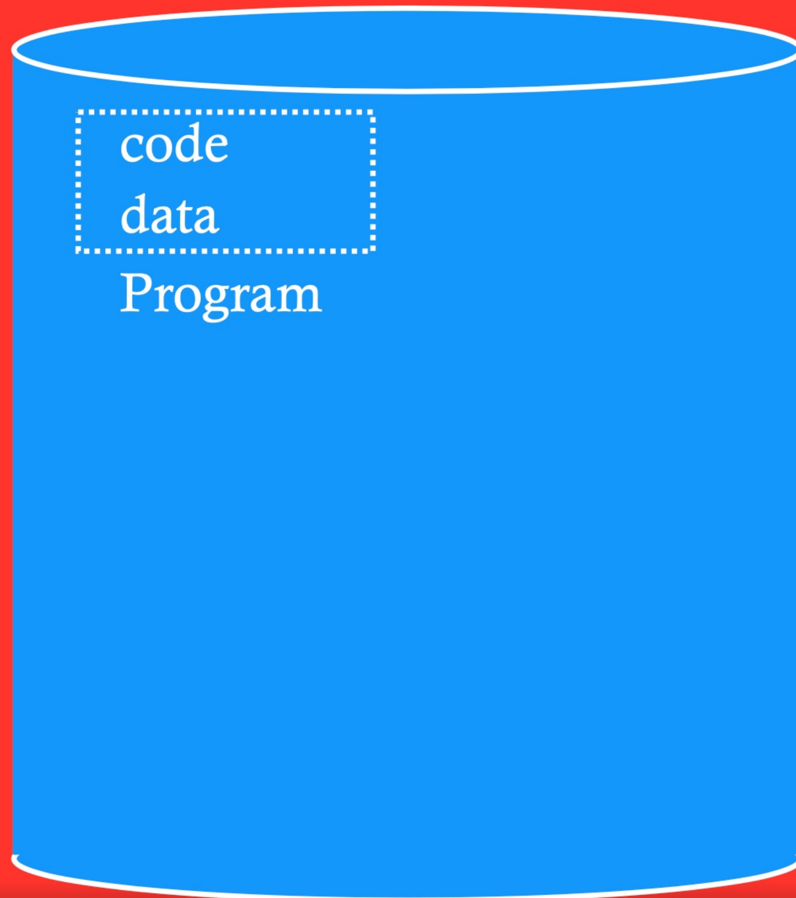
1. Time Sharing
2. Static Relocation
3. Base
4. Base+Bounds
5. Segmentation

1) Time Sharing of Memory

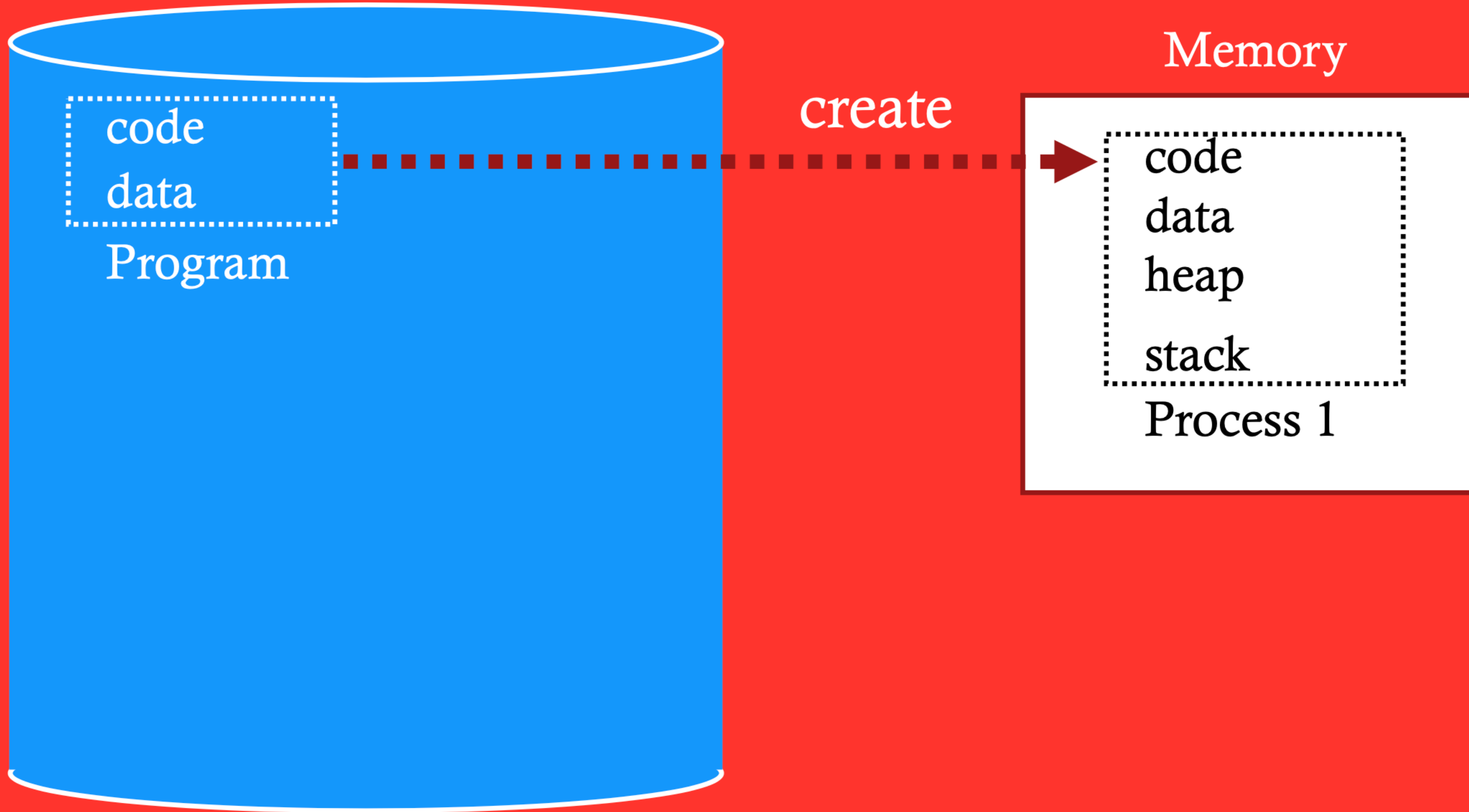
Try similar approach to how OS virtualizes CPU

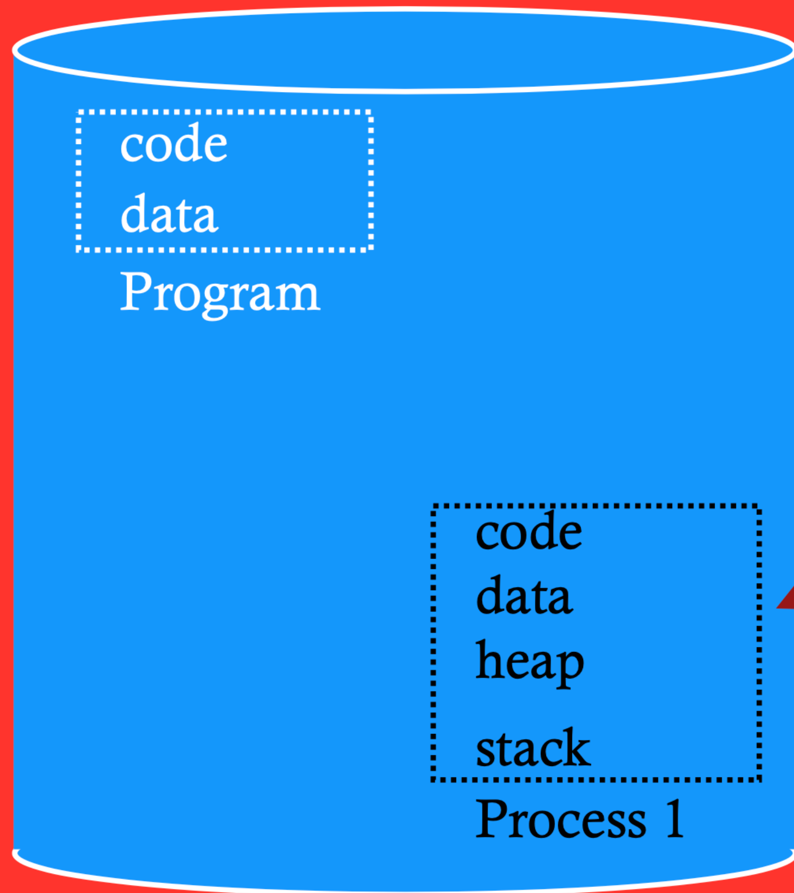
Observation:
OS gives illusion of many virtual CPUs by saving **CPU registers** to **memory**
when a process isn't running

Could give illusion of many virtual memories by saving **memory** to **disk**
when process isn't running

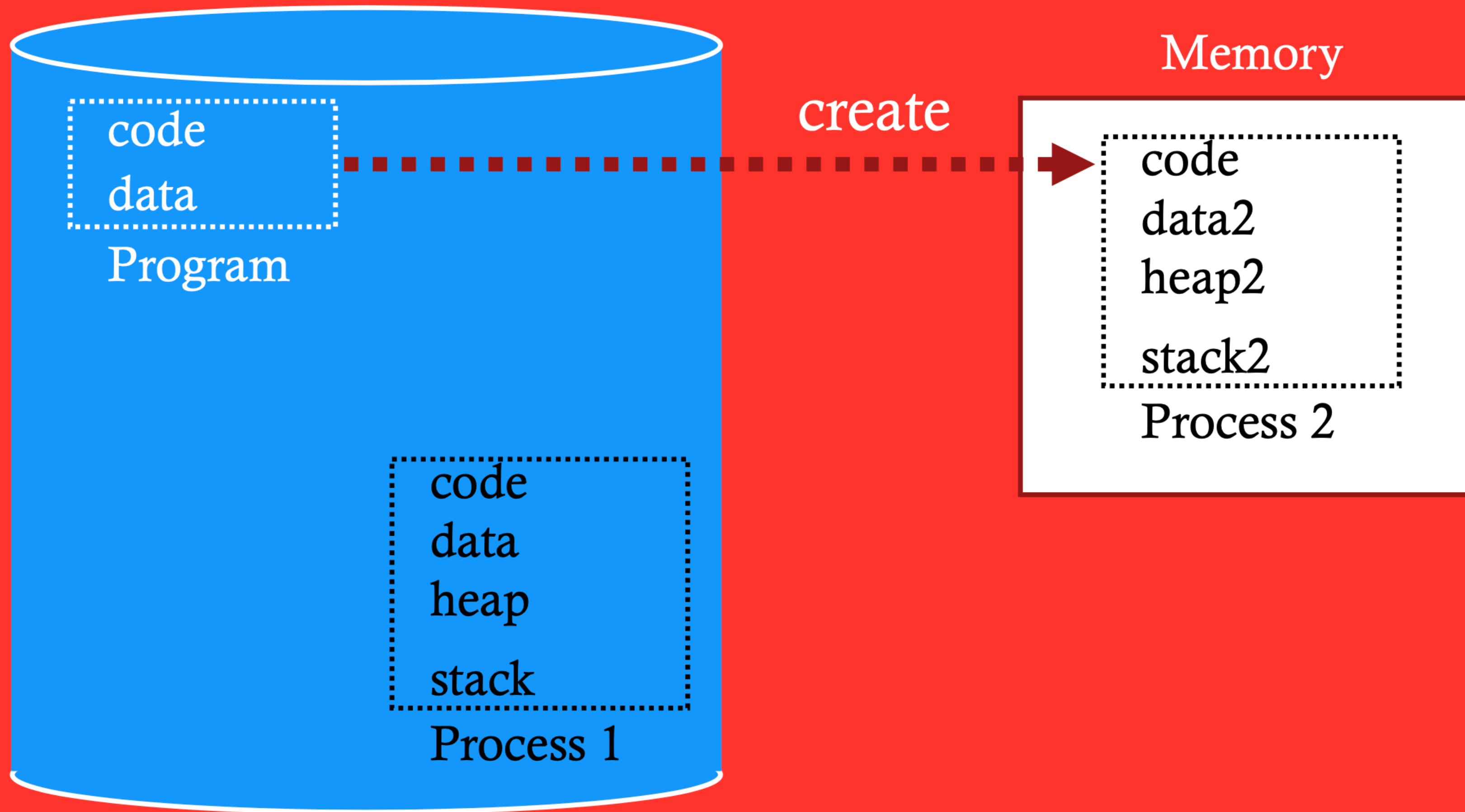


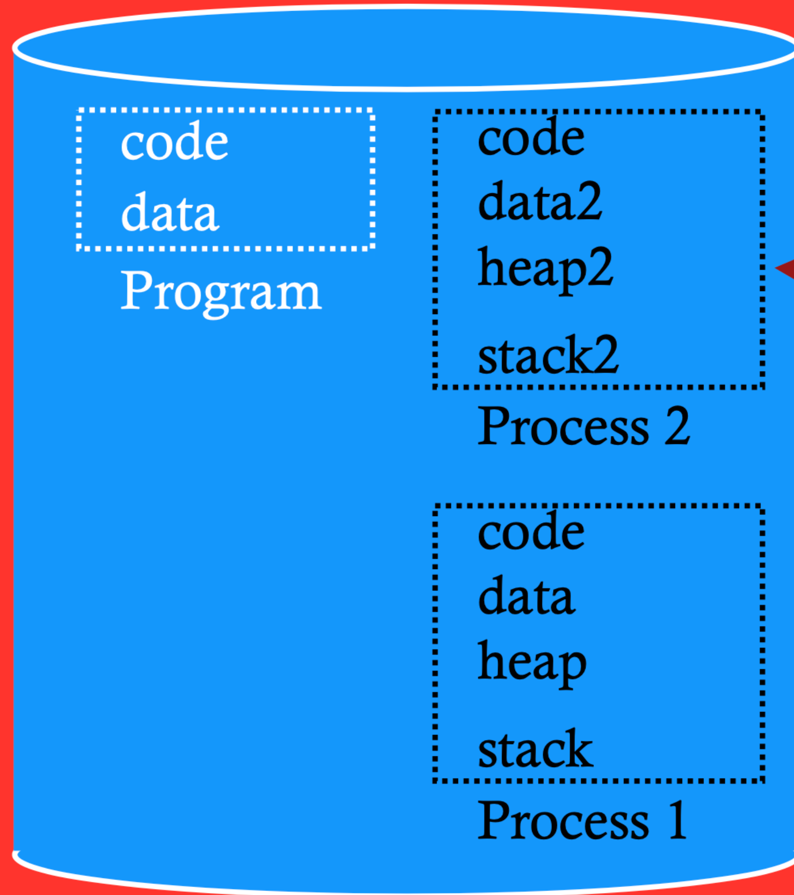
Time Share Memory: Example


code
data
Program
create
Memory
code
data
heap
stack
Process 1

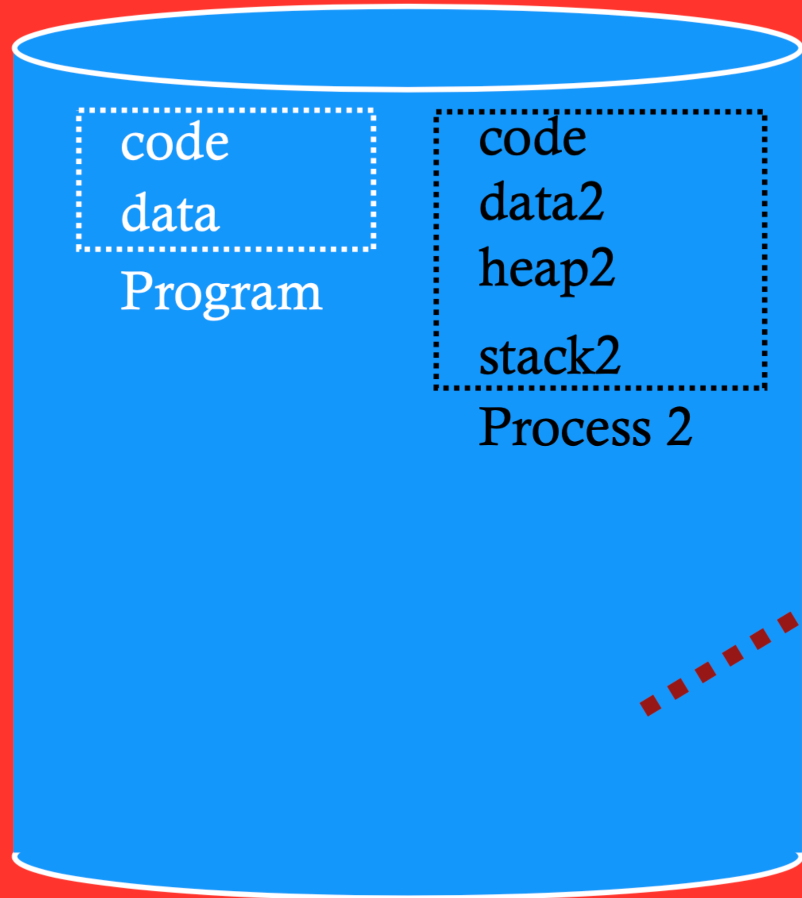

code
data
Program
code
data
heap
stack
Process 1

Memory

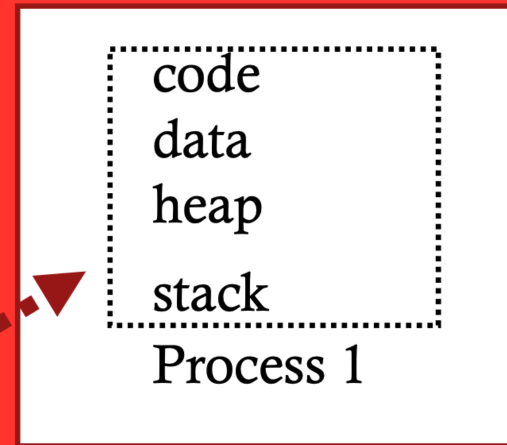

code
data
Program
create
Memory
code
data2
heap2
stack2
Process 2
code
data
heap
stack
Process 1

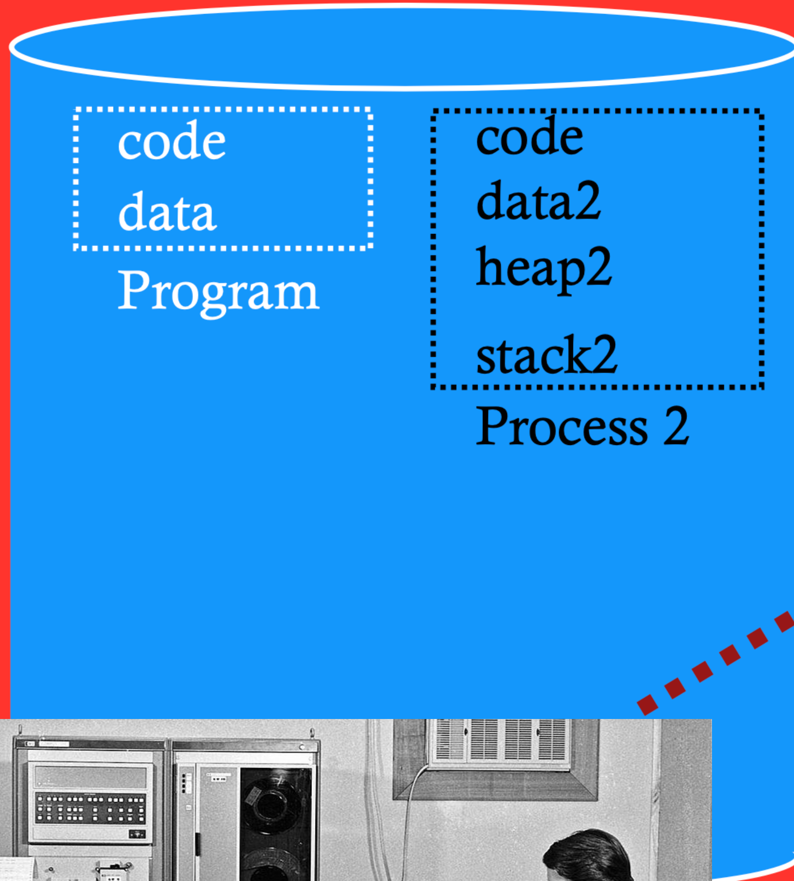

code
data
Program
code
data2
heap2
stack2
Process 2
code
data
heap
stack
Process 1

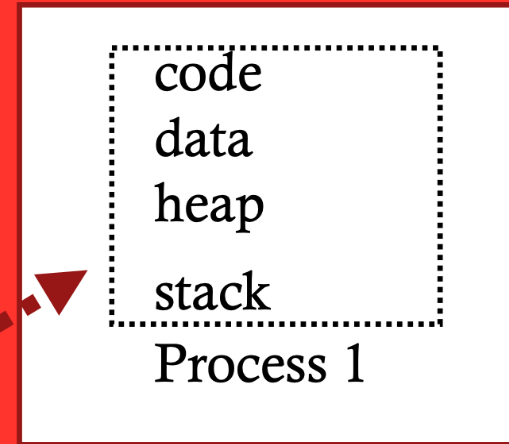
Memory



Memory




code
data
Program
code
data2
heap2
stack2
Process 2


Memory
code
data
heap
stack
Process 1

Problems with Time Sharing Memory

Problem: Ridiculously poor performance

Better Alternative: space sharing

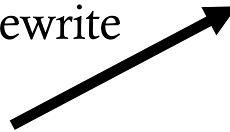
- At same time, space of memory is divided across processes

Remainder of solutions all use space sharing

2) Static Relocation

- Idea: OS rewrites each program before loading it as a process in memory
- Each rewrite for different process uses different addresses and pointers
- Change jumps, loads of static data

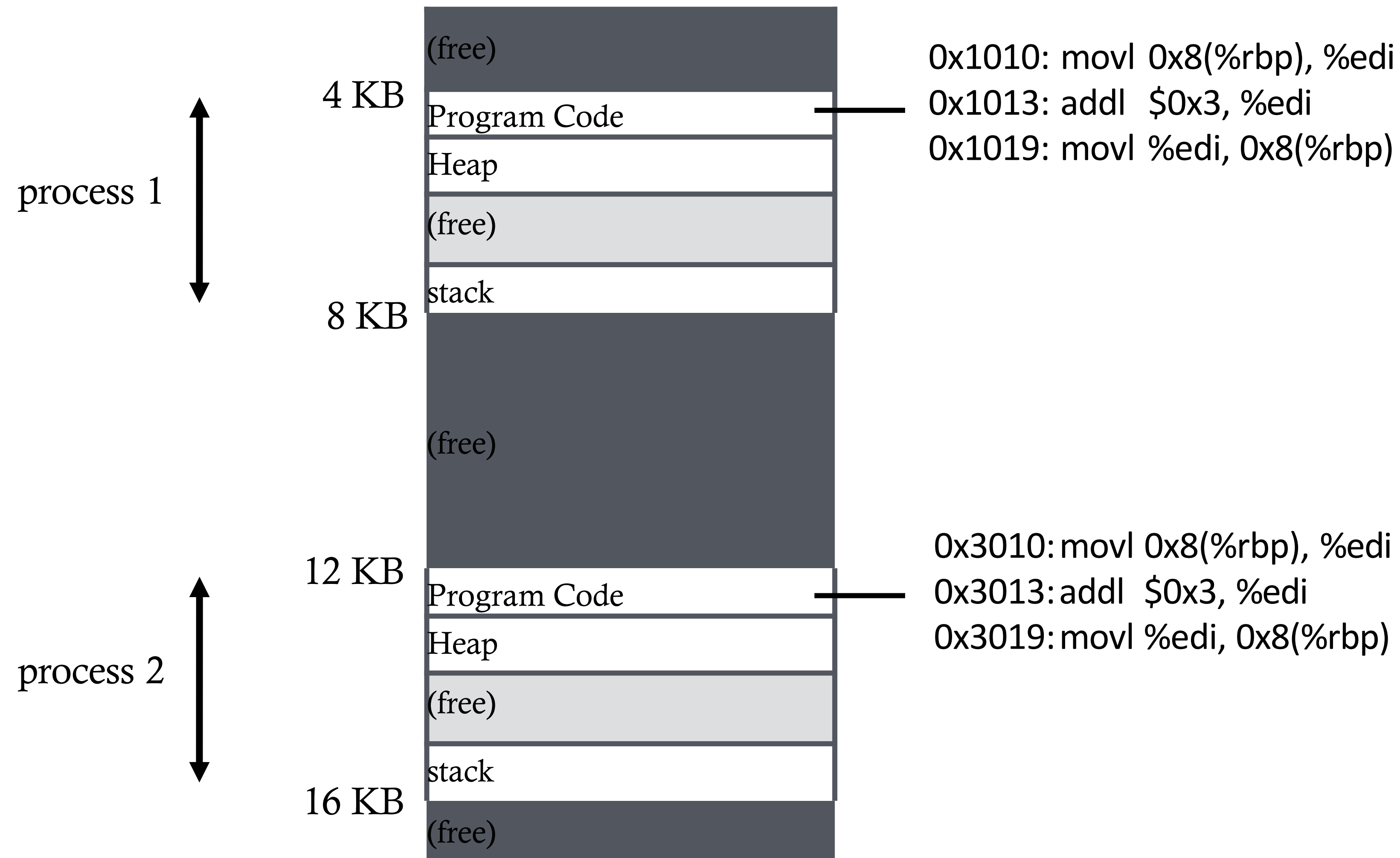
- 0x10: movl 0x8(%rbp), %edi
- 0x13: addl $0x3, %edi
- 0x19: movl %edi, 0x8(%rbp)



0x1010: movl 0x8(%rbp), %edi
0x1013: addl $0x3, %edi
0x1019: movl %edi, 0x8(%rbp)

0x3010: movl 0x8(%rbp), %edi
0x3013: addl $0x3, %edi
0x3019: movl %edi, 0x8(%rbp)

Static: Layout in Memory



Static Relocation: Disadvantages

No protection

- Process can destroy OS or other processes
- Possible to create addresses on the fly, and read/write
- No privacy

Cannot move address space after it has been placed

- May not be able to allocate new process

3) Dynamic Relocation

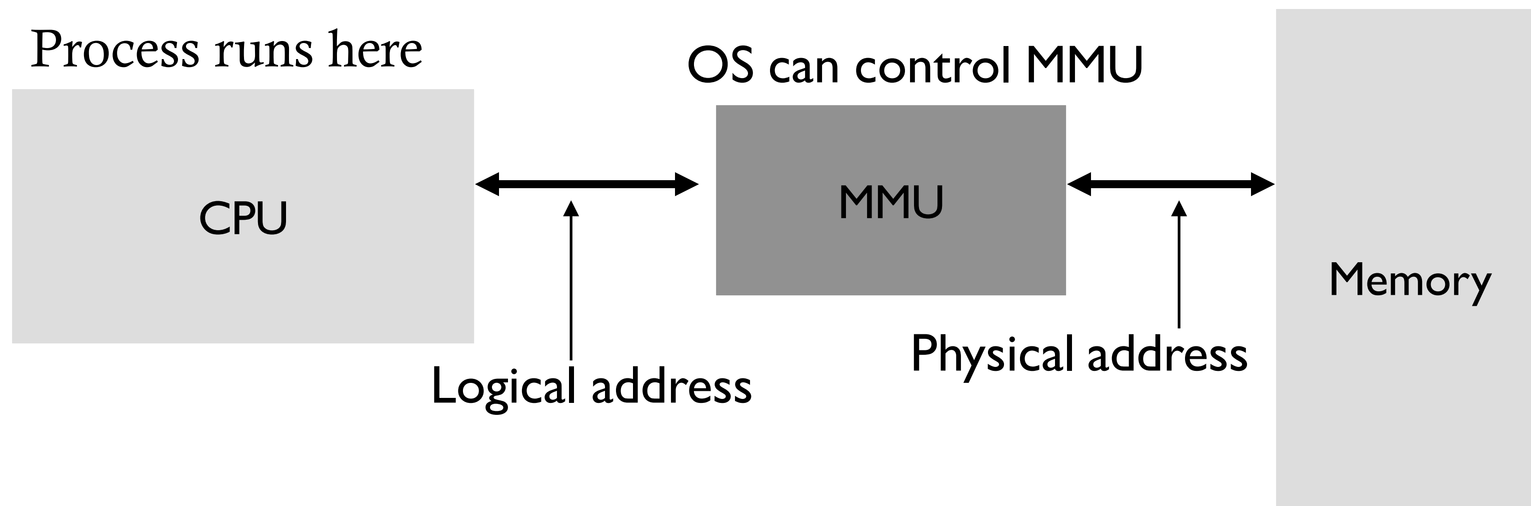
Goal: Protect processes from one another

Requires hardware support

- Memory Management Unit (MMU)

MMU dynamically changes process address at every memory reference

- Process generates logical or virtual addresses (in their address space)
- Memory hardware uses physical or real addresses



Hardware Support for Dynamic Relocation

Two operating modes

- Privileged (protected, kernel) mode: OS runs
  - When enter OS (trap, system calls, interrupts, exceptions)
  - Allows certain instructions to be executed
    - **Can manipulate contents of MMU**
  - **Allows OS to access all of physical memory**
- User mode: User processes run
  - **Perform translation of logical address to physical address**

Minimal MMU contains **base register** for translation

- base: start location for address space

Implementation of Dynamic Relocation: BASE REG

- Translation on every memory access of user process
  - MMU adds base register to logical address to form physical address

- MMU