Operating Systems 412

Pete Keleher

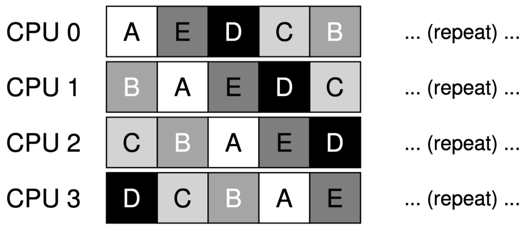
Scheduling

Multi-Processor Scheduling *SQMS*

- Simple approach is single-queue multiprocessor scheduling (SQMS)
  - each CPU simply grabs next job from queue
  - need synchronization (slow)
- Also: running process gains *affinity* for current CPU / core
  - registers
  - TLBs
  - *caches*
- Assume four cores, 5 CPUs:

Queue → A → B → C → D → E → NULL

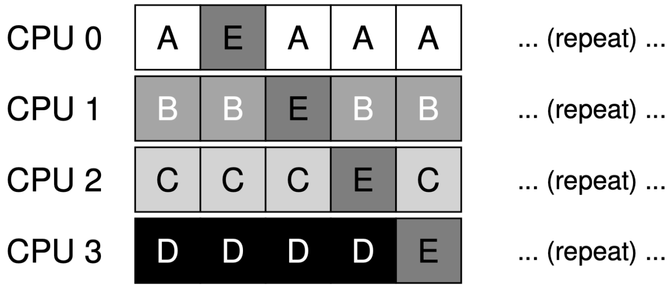
- Over time, might see:



CPU 0	A	E	D	C	B	... (repeat) ...
CPU 1	B	A	E	D	C	... (repeat) ...
CPU 2	C	B	A	E	D	... (repeat) ...
CPU 3	D	C	B	A	E	... (repeat) ...

Multi-Processor Scheduling *affinity*

- W/ affinity, might see:



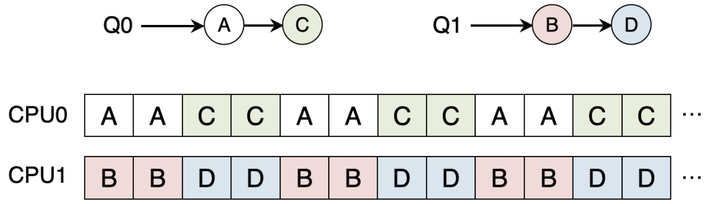
CPU 0	A	E	A	A	A	... (repeat) ...
CPU 1	B	B	E	B	B	... (repeat) ...
CPU 2	C	C	C	E	C	... (repeat) ...
CPU 3	D	D	D	D	E	... (repeat) ...

  - only *E* migrating among cores

- Even so, synchronization is bottleneck has #cores scales

Multi-Processor Scheduling *multi-queue scheduling*

- W/ round robin, might produce following schedule:



Q0 → A → C

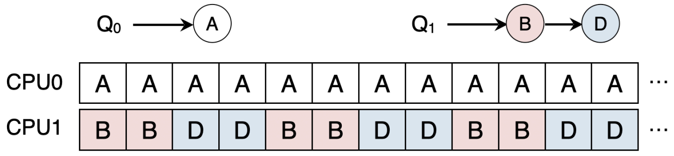
Q1 → B → D

CPU0	A	A	C	C	A	A	C	C	A	A	C	C	···
CPU1	B	B	D	D	B	B	D	D	B	B	D	D	···

- MQMS provides
  - scalability (especially for *embarassingly parallel* applications)
  - cache affinity

Multi-Queue Processor Scheduling *load imbalance*

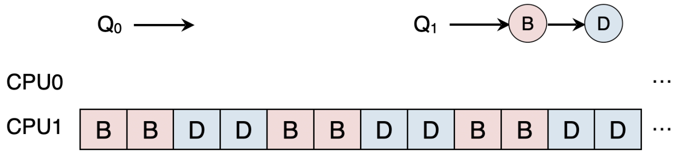
- After job C in Q_0 finishes:



CPU0	A	A	A	A	A	A	A	A	A	A	A	A	…
CPU1	B	B	D	D	B	B	D	D	B	B	D	D	…

A gets twice as much CPU as B and D.

- After job A in Q_0 finishes:

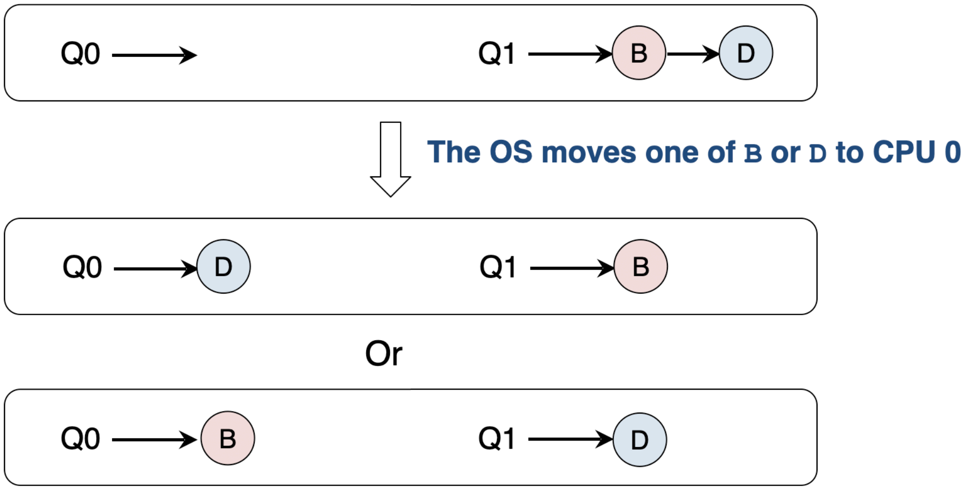


CPU0													…
CPU1	B	B	D	D	B	B	D	D	B	B	D	D	…

CPU0 will be left idle!

Multi-Queue Processor Scheduling *load imbalance*

- Migration:

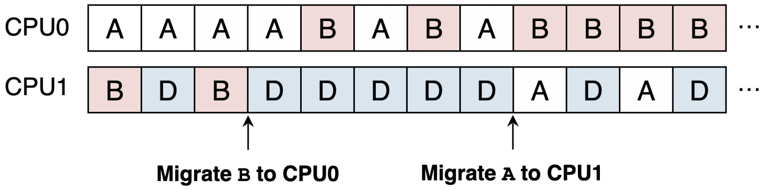


The OS moves one of B or D to CPU 0

Multi-Queue Processor Scheduling *load imbalance*

- Trickier case:

- Possible migration pattern:



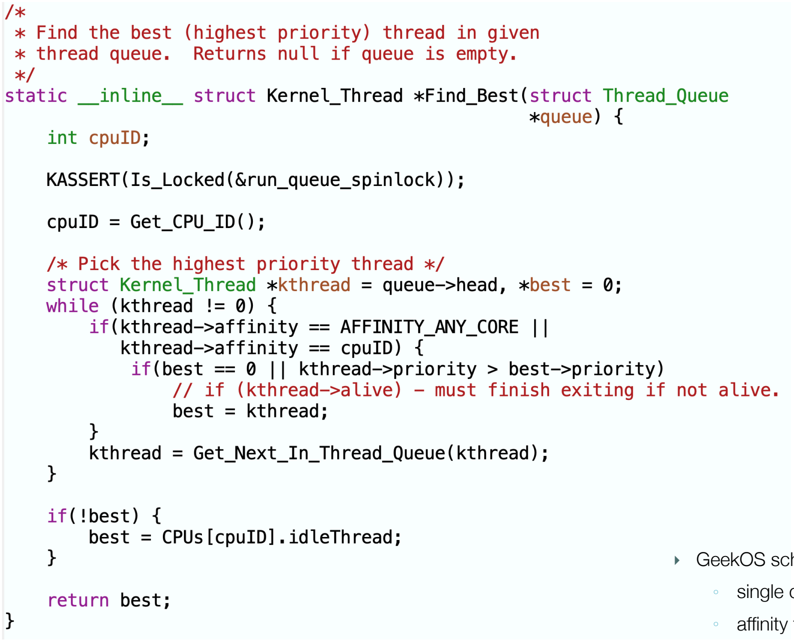
CPU0	A	A	A	A	B	A	B	A	B	B	B	B	…
CPU1	B	D	B	D	D	D	D	D	A	D	A	D	…

- Need to avoid flip-flopping

Multi-Queue Processor Scheduling *work stealing*

- Common approach is work stealing:
  - an underfull *source* queue peeks at other *target* queues
  - if target queue is more full than the source queue, it *steals* one or more jobs
- Issues
  - high overhead
  - problems scaling

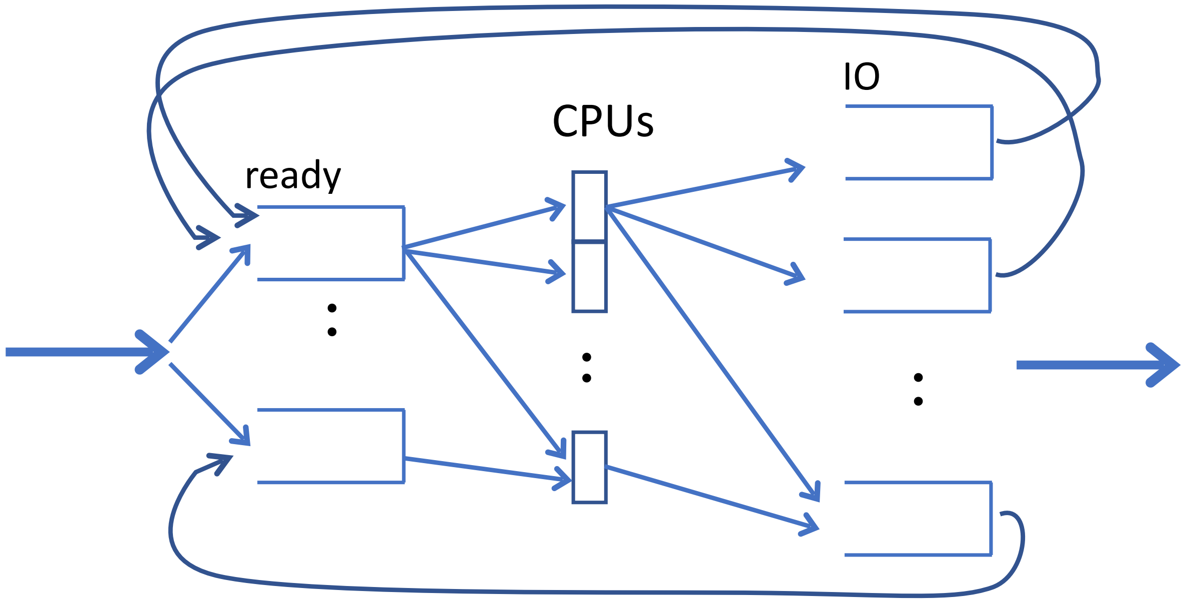
GeekOS *multi-core scheduler*



```
/*
 * Find the best (highest priority) thread in given
 * thread queue.  Returns null if queue is empty.
 */
static __inline__ struct Kernel_Thread *Find_Best(struct Thread_Queue
                                                  *queue) {
    int cpuID;

    KASSERT(Is_Locked(&run_queue_spinlock));

    cpuID = Get_CPU_ID();

    /* Pick the highest priority thread */
    struct Kernel_Thread *kthread = queue->head, *best = 0;
    while (kthread != 0) {
        if(kthread->affinity == AFFINITY_ANY_CORE ||
           kthread->affinity == cpuID) {
            if(best == 0 || kthread->priority > best->priority)
                // if (kthread->alive) - must finish exiting if not alive.
                best = kthread;
        }
        kthread = Get_Next_In_Thread_Queue(kthread);
    }

    if(!best) {
        best = CPUs[cpuID].idleThread;
    }

    return best;
}
```

- GeekOS scheduler
  - single queue
  - affinity for a specific CPU
  - searches for highest priority process w/ either:
    - no affinity for any CPU
    - affinity for the core doing the rescheduling

Queuing Theory *without probabilities*



Queuing Theory without probabilities

- Queueing system
  - servers + queues (waiting rooms)
  - customers arrive, wait, get served, depart or go to next server
  - queueing disciplines
    - non-preemptive: fifo, priority, …
    - preemptive: round-robin, multi-level feedback, ...

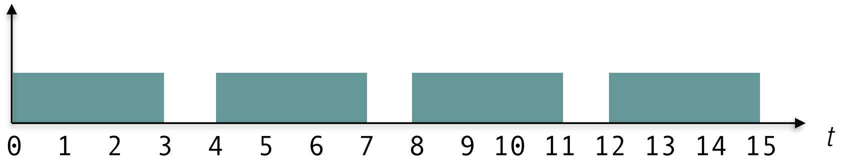
- Operating systems are examples of queueing systems
  - servers: hw/sw resources (cpu, disk, req handler, …)
  - customers: PCBs, TCBs, ...

- Given: arrival rates, service times, queueing disciplines, ...
- Obtain: queue sizes, response times, fairness, bottlenecks, ...

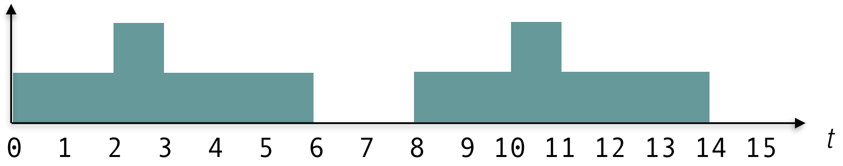
Queuing Theory without probabilities

- Consider cars traveling on a road with a turn
  - each car takes 3 seconds to go through the turn
  - at most one car can be in the turn at any time
- N(t): # cars in the turn and waiting to enter the turn

arrival rate 1/4
load 3/4
uniform



arrival rate 1/4
load 3/4
uniform



- Load < 1: *stable* w/ waits depending on burstiness
- Load > 1: *unstable*, ever-increasing waits

Queuing Theory without probabilities

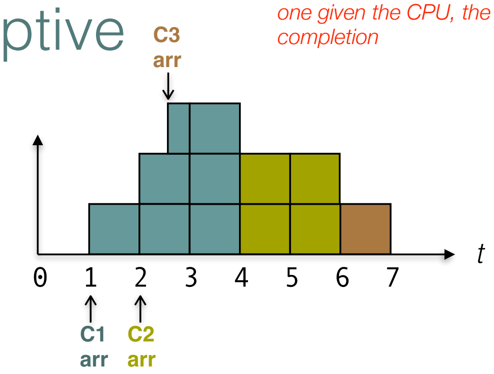
- Assume unending stream of customers:
  - arrival rate λ or X: *# arrivals per second*
  - average service time S: *work needed per customer*
  - average turnaround time R: *departure time D - arrival time A*
  - average wait time W: *turnaround time - service time*
  - throughput X: *# departures per sec averaged over all time*
  - average customers in system N: *waiting or busy*
  - utilization U: *fraction of time server is busy*
- Typical goal
  - Given: arrival rate, avg service time, queueing discipline
  - Obtain: average turnaround time, average queue size
- Little's Law (for any steady-state system):
  - $N = \lambda \times R$

FCFS non-preemptive

one given the CPU, the job runs to completion

customer	A_i	S_i	D_i	R_i	W_i
1	1.0	3.0	4.0	3.0	0.0
2	2.0	2.0	6.0	4.0	2.0
3	2.5	1.0	7.0	4.5	3.5

repeats every 10 seconds

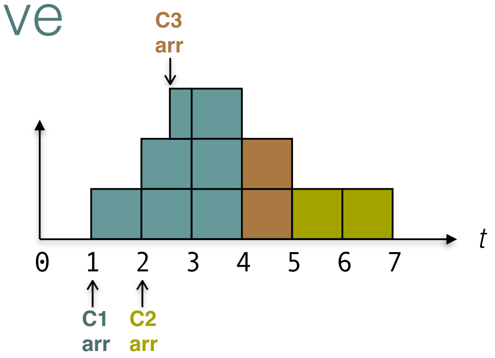


- System becomes empty at time 7 —> *stable*
  - Average turnaround time: $R = \frac{3.0 + 4.0 + 4.5}{3} = \frac{11.5}{3}$ sec
  - Average wait time: $W = \frac{0.0 + 2.0 + 3.5}{3} = \frac{5.5}{3}$ sec
  - Arrival rate = throughput: $\lambda = \frac{3}{10}$ arrivals / sec
  - Utilization: $U = \frac{6}{10}$
  - Average number customers: $N = \lambda \times R = \frac{3}{10} \times \frac{11.5}{3} = \frac{11.5}{10}$

SJF non-preemptive

customer	A_i	S_i	D_i	R_i	W_i
1	1.0	3.0	4.0	3.0	0.0
2	2.0	2.0	7.0	5.0	3.0
3	2.5	1.0	5.0	2.5	1.5

repeats every 10 seconds



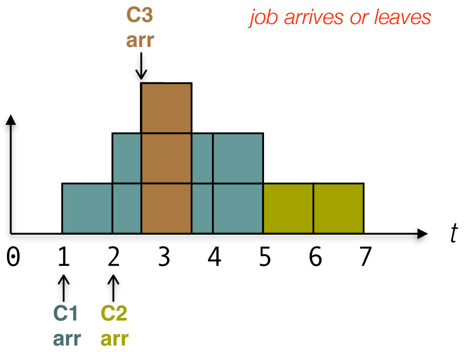
- System becomes empty at time 7 —> *stable*
  - Average turnaround time: $R = \frac{3.0 + 5.0 + 2.5}{3} = \frac{10.5}{3}$ sec
  - Average wait time: $W = \frac{0.0 + 3.0 + 1.5}{3} = \frac{4.5}{3}$ sec
  - Arrival rate = throughput: $\lambda = \frac{3}{10}$ arrivals/sec
  - Utilization: $U = \frac{6}{10}$
  - Average number customers: $N = \lambda \times R = \frac{3}{10} \times \frac{10.5}{3} = \frac{10.5}{10}$

SJS preemptive

assume can reschedule each time job arrives or leaves

customer	A_i	S_i	D_i	R_i	W_i
1	1.0	3.0	5.0	4.0	1.0
2	2.0	2.0	7.0	5.0	3.0
3	2.5	1.0	3.5	1.0	0.0

repeats every 10 seconds

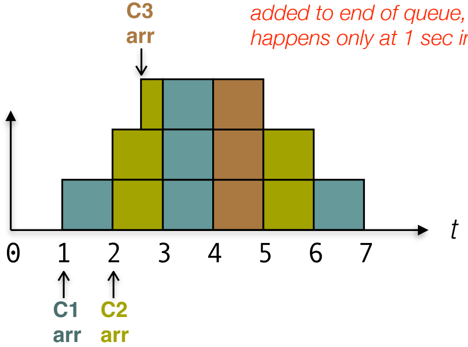


- System becomes empty at time 7 —> *stable*
  - Average turnaround time: $R = \frac{4.0 + 5.0 + 1.0}{3} = \frac{10.0}{3}$ sec
  - Average wait time: $W = \frac{1.0 + 3.0 + 0.0}{3} = \frac{4.0}{3}$ sec
  - Arrival rate = throughput: $\lambda = \frac{3}{10}$ arrivals / sec
  - Utilization: $U = \frac{6}{10}$
  - Average number customers: $N = \lambda \times R = \frac{3}{10} \times \frac{10.0}{3} = \frac{10}{10}$

RR preemptive

customer	A_i	S_i	D_i	R_i	W_i
1	1.0	3.0	7.0	6.0	3.0
2	2.0	2.0	6.0	4.0	2.0
3	2.5	1.0	5.0	2.5	1.5

repeats every 10 seconds



new job, or job finishing quantum, added to end of queue, scheduling happens only at 1 sec intervals

- System becomes empty at time 7 —> *stable*
  - Average turnaround time: $R = \frac{6.0 + 4.0 + 2.5}{3} = \frac{12.5}{3}$ sec
  - Average wait time: $W = \frac{3.0 + 2.0 + 1.5}{3} = \frac{6.5}{3}$ sec
  - Arrival rate = throughput: $\lambda = \frac{3}{10}$ arrivals / sec
  - Utilization: $U = \frac{6}{10}$
  - Average number customers: $N = \lambda \times R = \frac{3}{10} \times \frac{12.5}{3} = \frac{12.5}{10}$

Operating Systems 412

Pete Keleher

Memory

Memory

- 13 - Address Spaces
- 15 - Address Translation
- 16 - Segmentation
- 17 - Free Space Management
- 18 - Paging
- 19 - Translation Lookaside Buffers
- 20 - Advanced Paging
- 21 - Swapping
- 22 - Swapping Policy

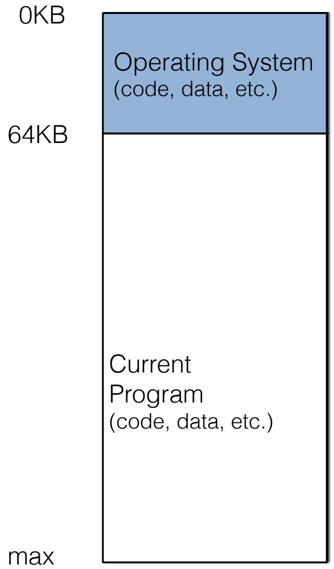
Memory Virtualization

- What is memory virtualization?
  - OS virtualizes its physical memory.
  - OS provides a virtual address space for each process.
  - Illusion of each process using the entire physical memory .

- Goals:
  - transparency
  - efficiency
    - in time *and* space
  - protection
    - for processes as well as OS

Early Operating Systems

- Load only one process in memory.
  - Poor utilization and efficiency



0KB

Operating System
(code, data, etc.)

64KB

Current
Program
(code, data, etc.)

max

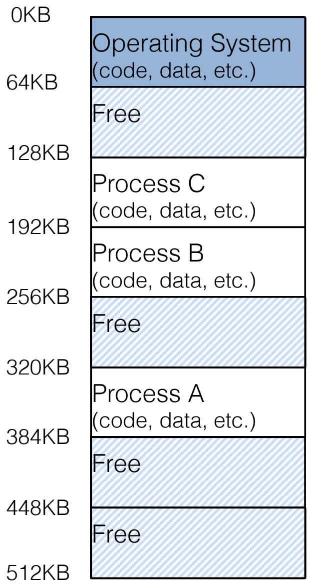
Physical Memory

Multiprogramming and Time Sharing

- Load multiple processes in memory
  - Execute one for a short while.
  - Switch processes between them in memory.
  - Better utilization and efficiency.

- But what about protection?
  - Errant memory accesses from other processes

- Also:
  - fragmentation
  - shared libraries
  - not efficient if we have many small processes



0KB

Operating System
(code, data, etc.)

64KB

Free

128KB

Process C
(code, data, etc.)

192KB

Process B
(code, data, etc.)

256KB

Free

320KB

Process A
(code, data, etc.)

384KB

Free

448KB

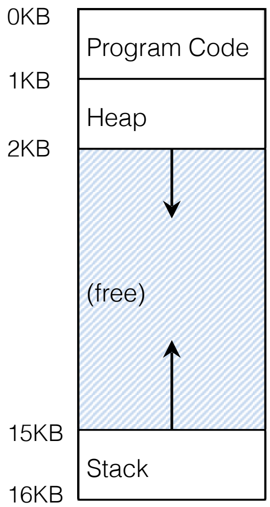
Free

512KB

Physical Memory

Address Space

- An abstraction of physical memory:



Address Space

- Code
  - Where instructions live
- Heap
  - Dynamically allocate memory.
    - `malloc` in C
    - `new` in object-oriented languages
- Stack
  - Store return addresses or values.
  - Contain local variables arguments to routines.

Virtual Addresses

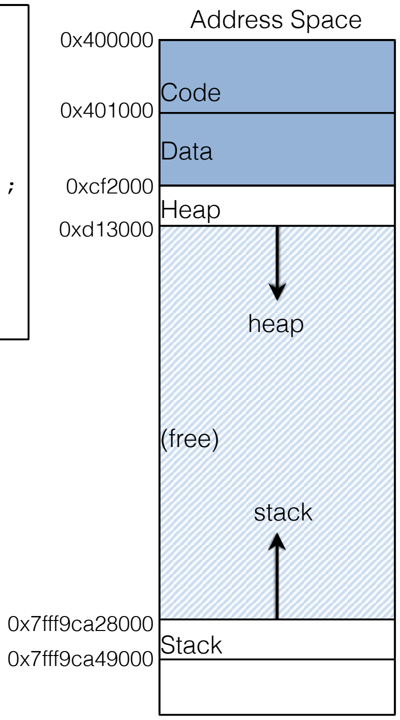
- Every address in a running program is virtual.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){

    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```

- OS uses hardware to translate virtual addresses to physical

Virtual Addresses

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){

    printf("location of code  : %p\n", (void *) main);
    printf("location of heap  : %p\n", (void *) malloc(1));
    int x = 3;
    printf("location of stack : %p\n", (void *) &x);

    return x;
}
```



Address Space

0x400000

Code

0x401000

Data

0xcf2000

Heap

0xd13000

heap

(free)

stack

0x7fff9ca28000

Stack

0x7fff9ca49000

Output in 64-bit Linux machine:

```
location of code  : 0x40057d
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```

Need Efficiency, and Control…

- Remember: Limited direct execution (LDE)
  - Programs run directly (not emulated)
  - Memory virtualizing, efficiency, control maintained by hardware support.
    - e.g., registers, TLBs (Translation Look-aside Buffers), page-tables
- Hardware transforms virtual addresses to physical addresses
  - Memory only addressed with physical addresses
- The OS sets up the hardware.
  - Hardware raises interrupts when needed.

Example: Address Translation

```
void func()
        int x;
        ...
        x = x + 3; // this is the line of code we are interested in
```

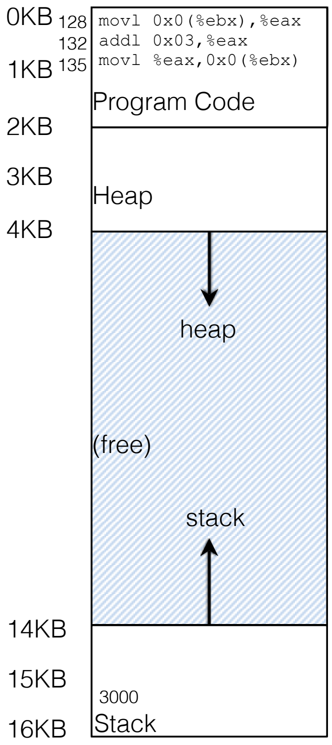
- **Load** a value from memory
- **Increment** by three
- **Store** the value back into memory

- Assembly

```
128 : movl 0x0(%ebx), %eax        ; load 0+ebx into eax
132 : addl $0x03, %eax            ; add 3 to eax register
135 : movl %eax, 0x0(%ebx)        ; store eax back to mem
```

- Assume address of `'x'` in `ebx` register.
- **Load** the value at that address into `eax` register.
- **Add** 3 to `eax` register.
- **Store** the value in `eax` back into memory.

Example: Address Translation

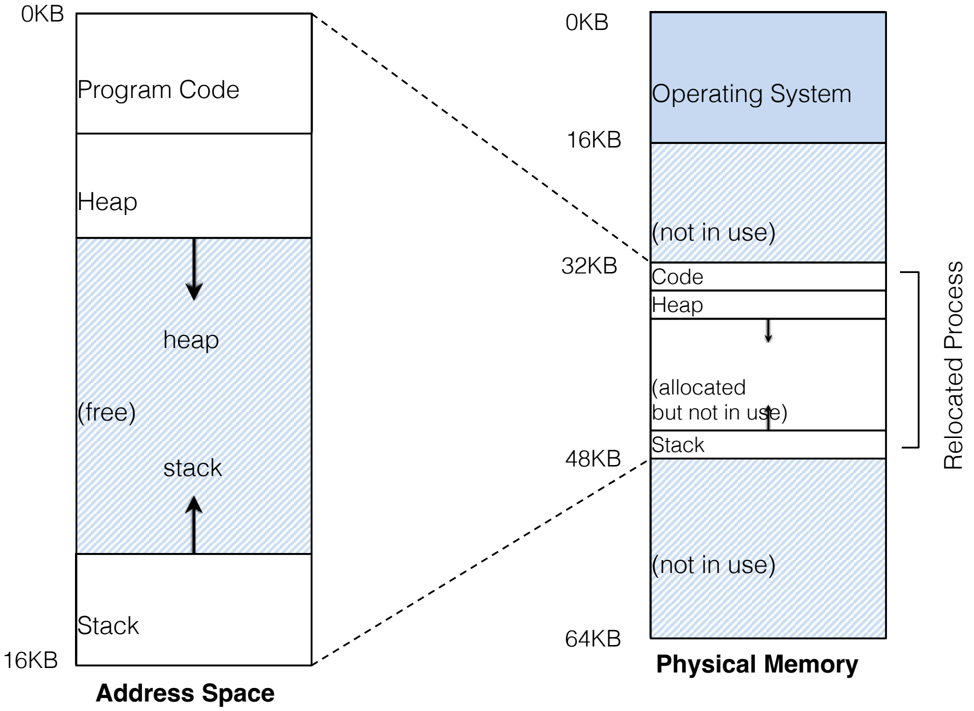


```
128 movl 0x0(%ebx),%eax
132 addl 0x03,%eax
135 movl %eax,0x0(%ebx)
```

- Fetch instruction at address 128
- Execute instruction (load from address 15KB)
- Fetch instruction at address 132
- Execute instruction (no memory reference)
- Fetch the instruction at address 135
- Execute instruction (store to address 15 KB)

But not all programs can be at location 0

A Single Relocated Process



One Approach *base and bounds*