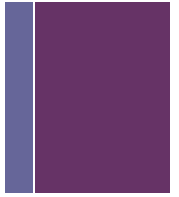




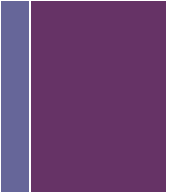
Final Exam

# + Logistics



- **100 minutes (half an hour longer than midterm)**
  - Covers all materials, emphasizes second half
  - You may bring 1 page (8.5 by 11, two-sided) of notes. It must be hand-written.
- **Disclaimer: this review is not complete.**
  - Not all exam materials are mentioned by this review!
  - We are not even mentioning things from the first half of the semester.

# + General Tips



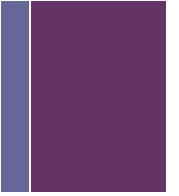
- Read through all questions, do what you think are the easier ones first.
- Make sure you have *completed and understood* all the homeworks, recitation exercises, labs and exams.
- Use Piazza! Have discussions there with your classmates.



+

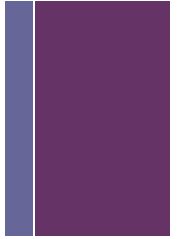
Machine-Level Programming

# + Machine-Level Programs



- **Basic information about x84-64**
- **Source, assembly, object & machine code**
- **General purpose registers**
- **Data movement instructions.**
  - `movq %rdi, %rsi`
  - `movq (%rdi), %rsi`

# + Memory Addressing



- General form

**$D(Rb, Ri, S)$                        $Mem[ D + Reg[Rb] + Reg[Ri] * S ]$**

- **D:**     Constant “displacement”
  - **Rb:**    Base register: Any of 16 integer registers
  - **Ri:**    Index register: Any, except for **%rsp**
  - **S:**     Scale: 1, 2, 4, or 8
- Special cases: you can omit certain arguments if not needed.

**$(Rb, Ri)$                        $Mem[ Reg[Rb] + Reg[Ri] ]$**

**$D(Rb, Ri)$                        $Mem[ D + Reg[Rb] + Reg[Ri] ]$**

**$(Rb, Ri, S)$                        $Mem[ Reg[Rb] + Reg[Ri] * S ]$**

# + Address Computation

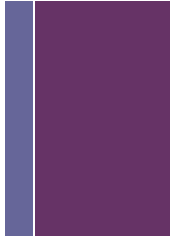
- `leaq src, dest`
  - `src` is an address computation expression
  - set `dest` to address denoted by expression
- use case 1
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
- Example

```
char* a2(char* x){  
    return &x[2];  
}
```

```
leaq 2(%rdi), %rax    # return &x[2]  
ret
```



# + Arithmetic Operations



- Two Operand Instructions:

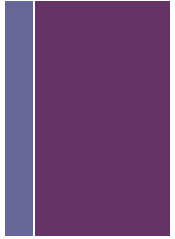
<u>Format</u>	<u>Computation</u>
▪ <b>addq</b> src, dest	dest = dest + src
▪ <b>subq</b> src, dest	dest = dest - src
▪ <b>imulq</b> src, dest	dest = dest * src
▪ <b>salq</b> src, dest	dest = dest << src (also called <b>shlq</b> )
▪ <b>sarq</b> src, dest	dest = dest >> src (arithmetic)
▪ <b>shrq</b> src, dest	dest = dest >> src (logical)
▪ <b>xorg</b> src, dest	dest = dest ^ src
▪ <b>andq</b> src, dest	dest = dest & src
▪ <b>orq</b> src, dest	dest = dest   src

- Watch out for argument order!





# Arithmetic Expression Example



```
long arith (long x, long y, long z){  
    long t1 = x + y;  
    long t2 = z + t1;  
    long t3 = x + 4;  
    long t4 = y * 48;  
    long t5 = t3 + t4;  
    long rval = t2 * t5;  
    return rval;  
}
```

```
arith:  
    leaq    (%rdi,%rsi), %rax    #t1  
    addq    %rdx, %rax          #t2  
    leaq    (%rsi,%rsi,2), %rdx  
    salq    $4, %rdx            #t4  
    leaq    4(%rdi,%rdx), %rcx   #t5  
    imulq   %rcx, %rax           #rval  
    ret
```

- Noteworthy instructions:
  - **leaq**: “address” computation
  - **salq**: shift
  - **imulq**: integer multiplication

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b>
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

# + Control

- Information about currently executing program...
  - temporary data  
( `%rax`, ... )
  - location of runtime stack  
( `%rsp` )
  - location of current code point  
( `%rip` )
  - status of recent tests  
( `CF`, `ZF`, `SF`, `OF` )

## Registers

<code>%rax</code>	<code>%r8</code>
<code>%rbx</code>	<code>%r9</code>
<code>%rcx</code>	<code>%r10</code>
<code>%rdx</code>	<code>%r11</code>
<code>%rsi</code>	<code>%r12</code>
<code>%rdi</code>	<code>%r13</code>
<code>%rsp</code>	<code>%r14</code>
<code>%rbp</code>	<code>%r15</code>

`%rip`

## Instruction

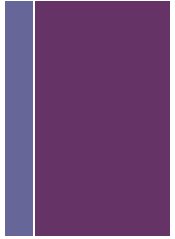
<code>CF</code>	<code>ZF</code>	<code>SF</code>	<code>OF</code>
-----------------	-----------------	-----------------	-----------------

Condition

Current stack 'top'    Current instruction



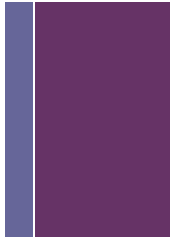
# Condition Codes (Implicit Setting)



- **Single bit registers**
  - **CF** Carry Flag (for unsigned)
  - **SF** Sign Flag (for signed)
  - **ZF** Zero Flag
  - **OF** Overflow Flag (for signed)
- **Implicitly set (think of it as a side effect) by arithmetic operations**
  - Example: **addq *src, dest***  $\leftrightarrow b = a + b$ 
    - **CF** set if carry out from most significant bit (unsigned overflow)
    - **ZF** set if  $t == 0$
    - **SF** set if  $t < 0$  (as signed)
    - **OF** set if two's-complement (signed) overflow  
 $(a > 0 \ \&\& \ b > 0 \ \&\& \ t < 0) \ || \ (a < 0 \ \&\& \ b < 0 \ \&\& \ t > 0)$
- **Not set by leaq instruction (!!!)**



# Conditional Branching by Jumping



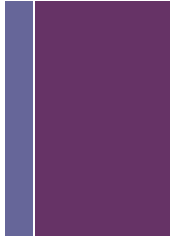
```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x - y;
    else
        result = y - x;
    return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi # y, x
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:      # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rax	Return value



# Conditional Move Example



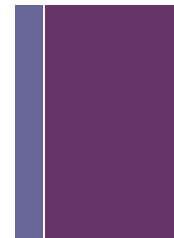
```
long absdiff(long x, long y){  
    long result;  
    if (x > y)  
        result = x-y;  
    else  
        result = y-x;  
    return result;  
}
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rax	Return value
%rdx	Temp variable

```
absdiff:  
    movq    %rdi, %rax    # x  
    subq    %rsi, %rax    # if-val = x-y  
    movq    %rsi, %rdx  
    subq    %rdi, %rdx    # else-val = y-x  
    cmpq    %rsi, %rdi    # %rsi = y, %rdi = x  
    cmovle  %rdx, %rax    # if y <= x, result = else-val  
    ret
```



# “Do-While” Loop Compilation



```
long pcount_goto(unsigned long x){
    long result = 0;
loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}
```

Register	Use(s)
%rdi	Argument <b>x</b>
%rax	<b>result</b>

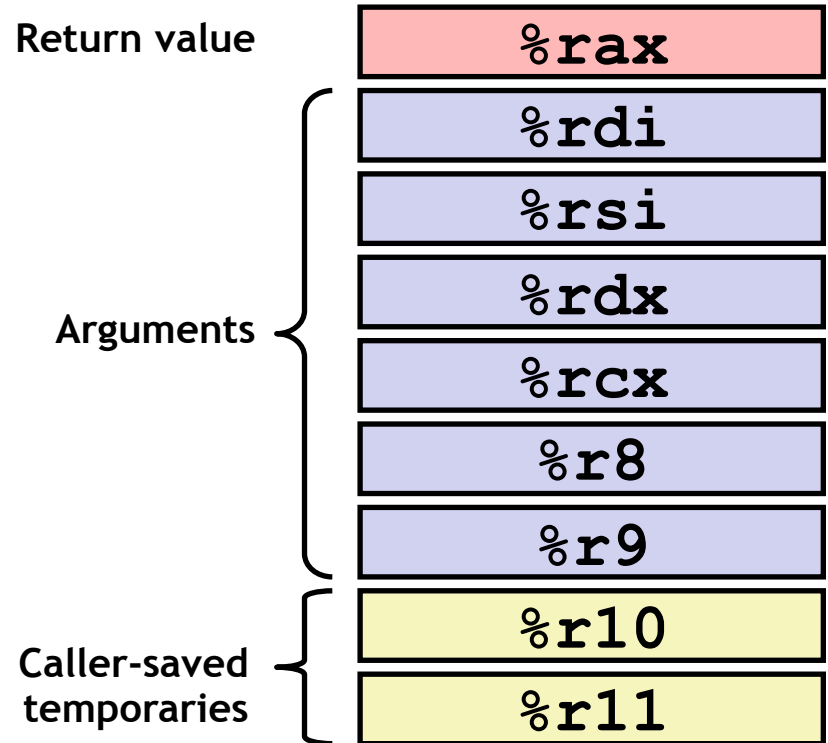
```
    movl    $0, %rax           # result = 0
.L2:                                     # loop:
    movq    %rdi, %rdx
    andq    $1, %rdx           # t = x & 0x1
    addq    %rdx, %rax         # result += t
    shrq    %rdi               # x >>= 1
    jne     .L2                # if (x) goto loop
    rep; ret
```

- Note: some processors' branch predictors behave badly when a branch's target or fall-through is a **ret** instruction, and adding the **rep;** prefix avoids this.

# + x86-64 Caller-saved Registers



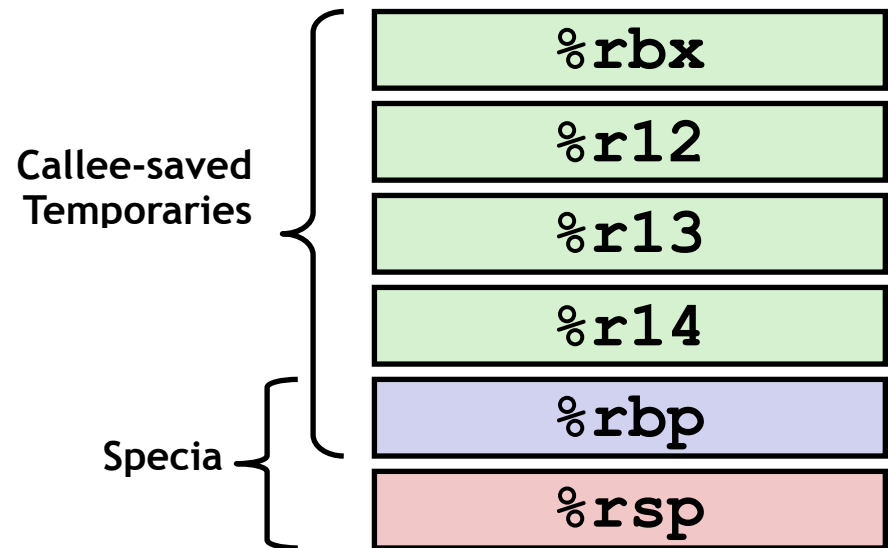
- **%rax**
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- **%rdi, ..., %r9**
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- **%r10, %r11**
  - Caller-saved
  - Can be modified by procedure



# + x86-64 Callee-saved Registers



- **%rbx, %r12, %r13, %r14**
  - Callee-saved
  - Callee must save & restore
- **%rbp**
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
- **%rsp**
  - Special form of callee save
  - Restored to original value upon exit from procedure

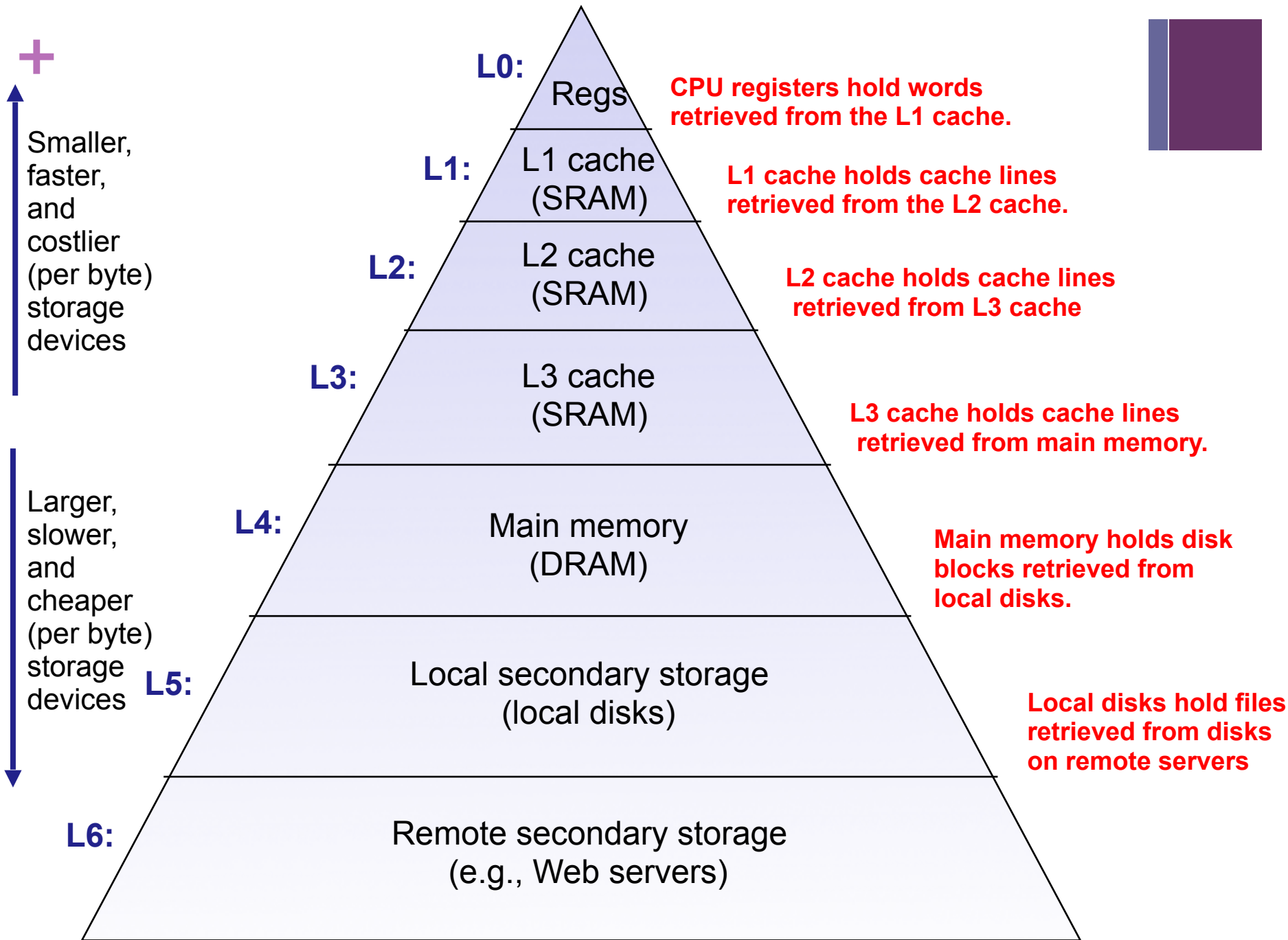






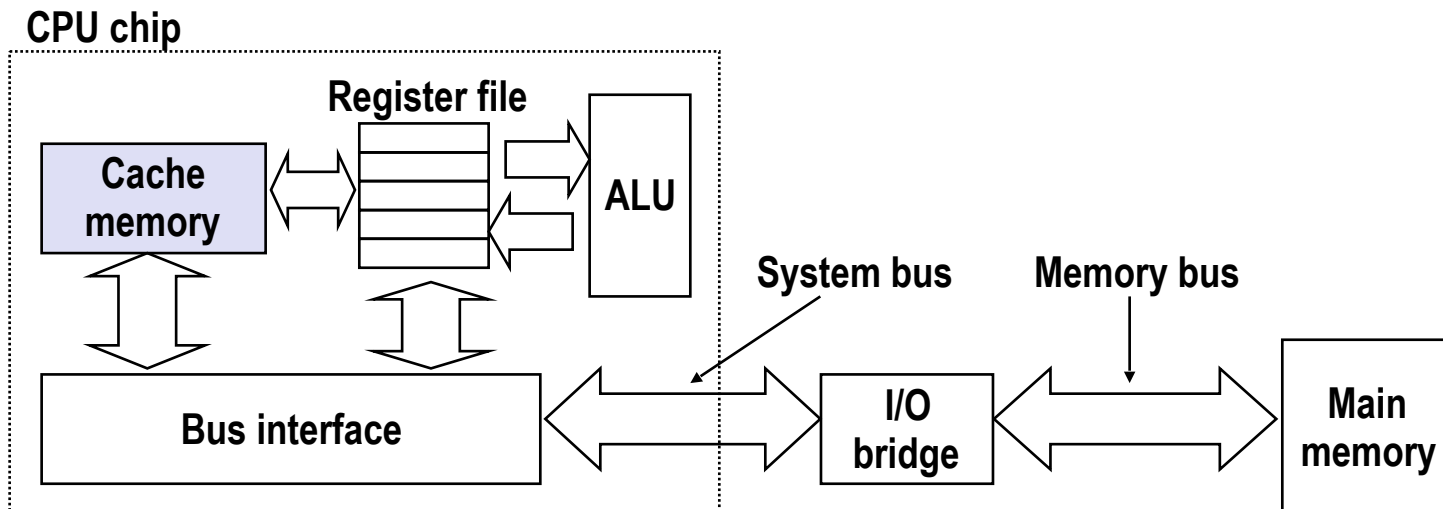
+

Memory



# + Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:



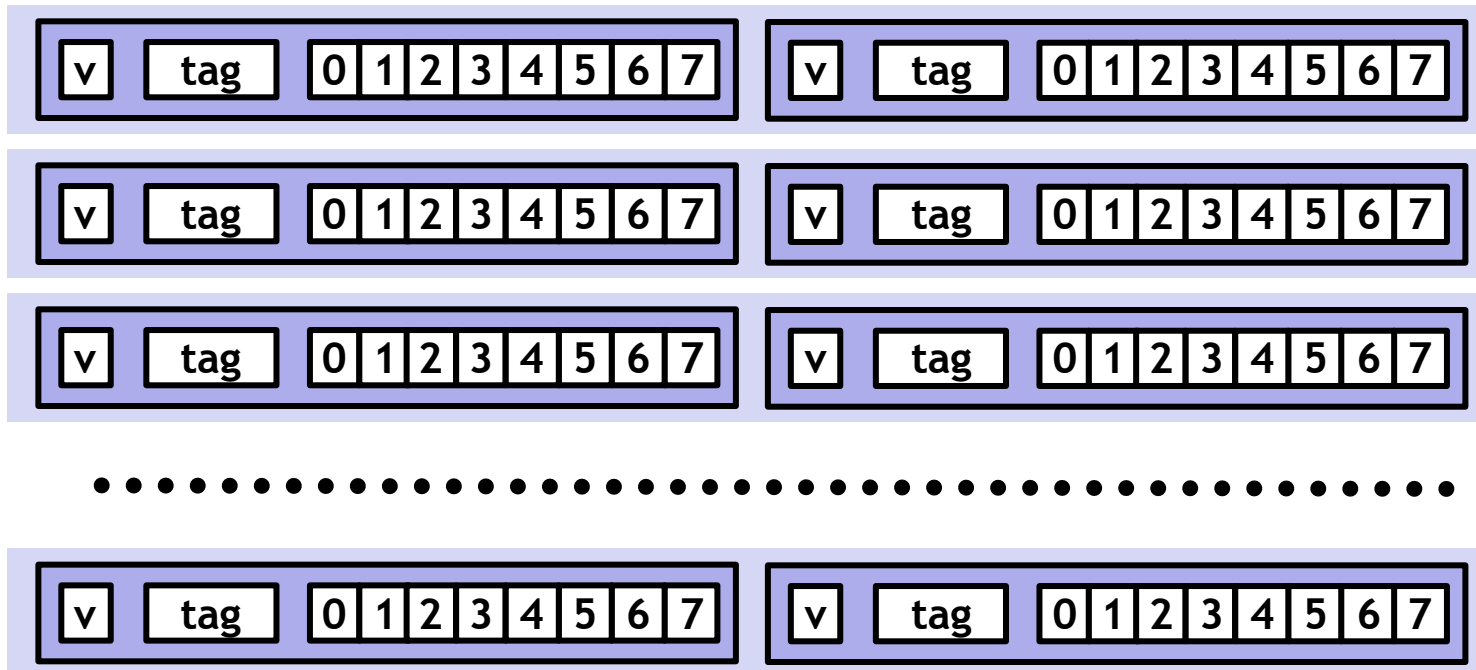
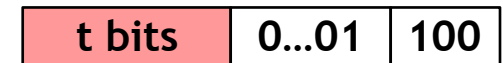
# + E-way Set Associative Cache (E = 2)



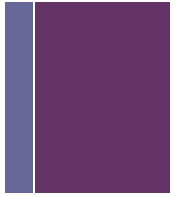
E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of short int:



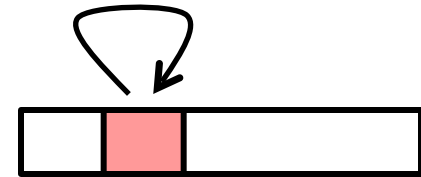
# + Locality



- **Principle of Locality:** Programs tend to use data and instructions with addresses near or equal to those they have used recently

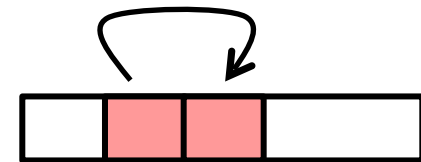
- **Temporal locality:**

- Recently referenced items are likely to be referenced again in the near future

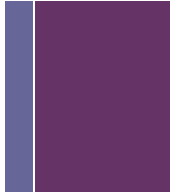


- **Spatial locality:**

- Items with nearby addresses tend to be referenced close together in time

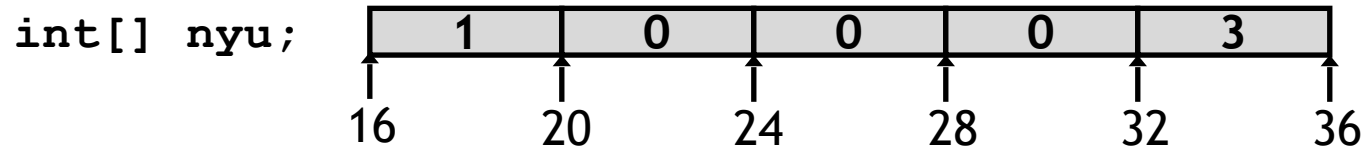


# + Stride-n Access Patterns



- **C arrays allocated in row-major order, contiguously**
- **Stepping through columns in one row:**
  - `for (i = 0; i < N; i++)`  
    `sum += a[0][i];`
  - accesses successive, contiguous elements
- **Stepping through rows in one column:**
  - `for (i = 0; i < n; i++)`  
    `sum += a[i][0];`
  - accesses distant elements (stride-rowsize pattern)
  - no spatial locality!

# + Array Accessing Example

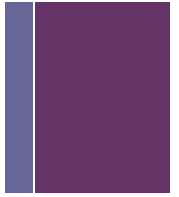


```
int get_digit(int[] z, int digit){  
    return z[digit];  
}
```

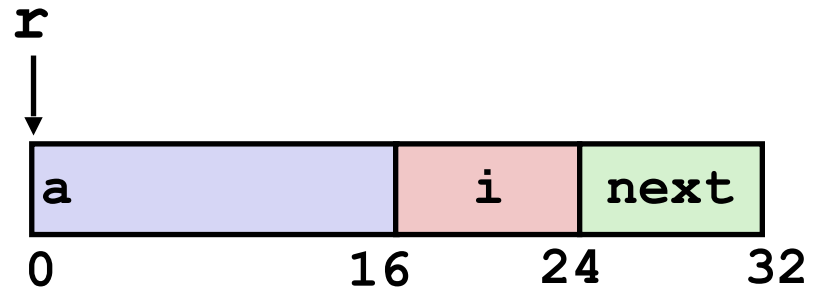
```
# %rdi = z  
# %rsi = digit  
movq (%rdi,%rsi,4), %rax # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi + 4*%rsi`
- Use memory reference `(%rdi,%rsi,4)`

# + Structure Representation



```
struct rec {  
    int a[4];  
    long i;  
    struct rec *next;  
};
```

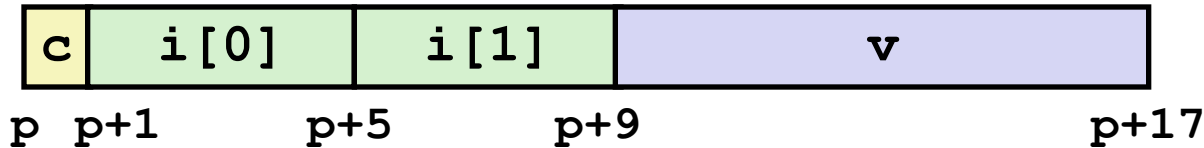


- **Structure represented as block of memory**
  - Big enough to hold all of the fields
- **Fields ordered according to declaration**
  - Even if another ordering could yield a more compact representation
  - Machine-level program has no understanding of the structures in the source code



# + Structures & Alignment

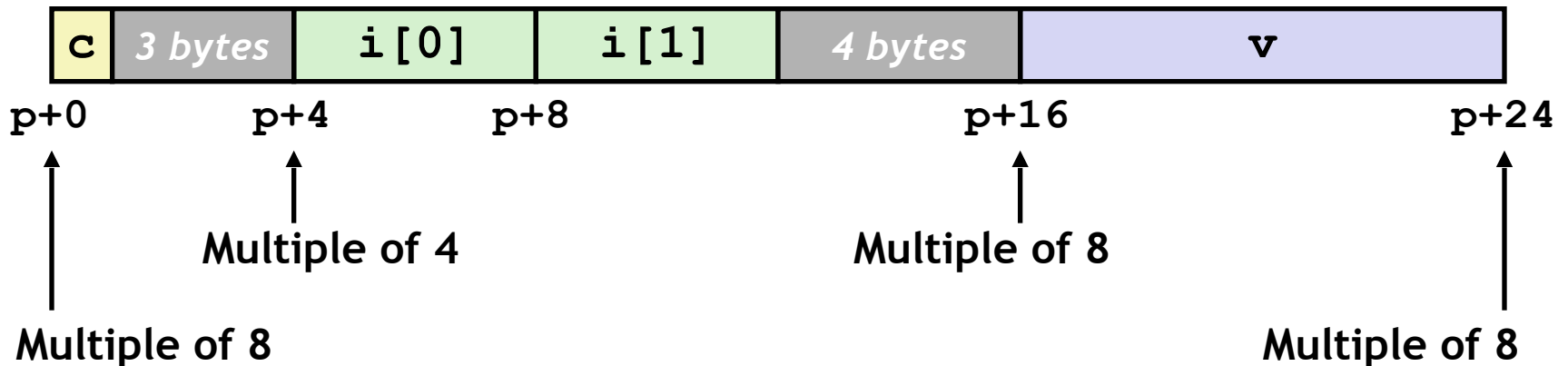
## ■ Unaligned Data



```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```

## ■ Aligned Data

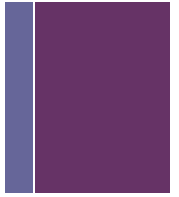
- Primitive data type requires  $K$  bytes
- Address must be multiple of  $K$



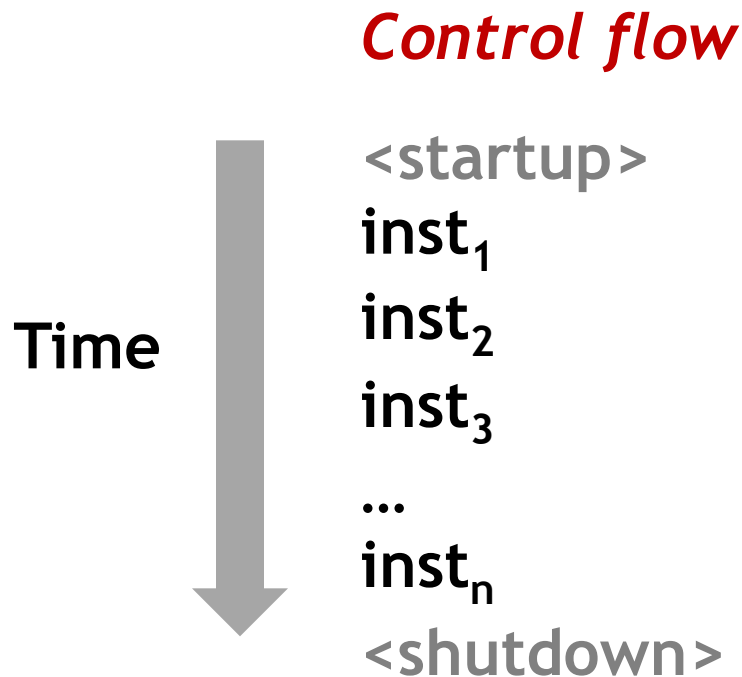


# Exceptions & Processes

# + Control Flow

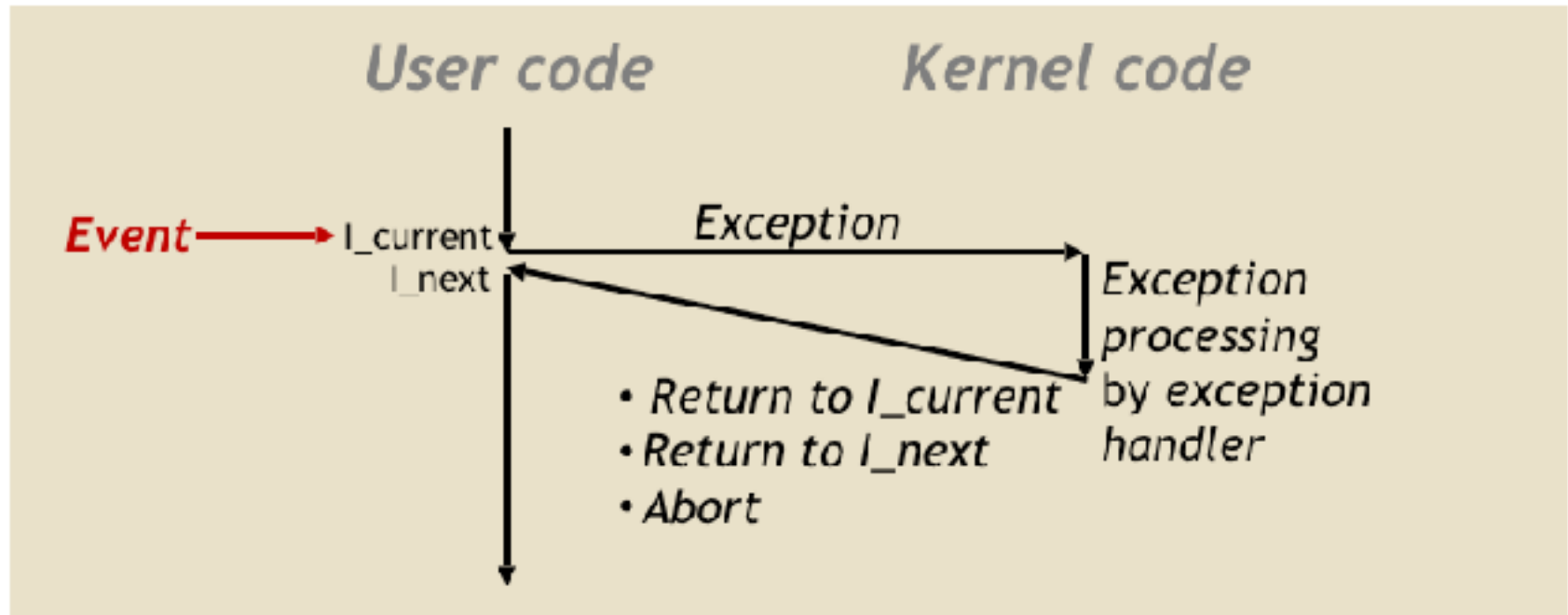


- **Processors do only one thing:**
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's control flow (or flow of control)

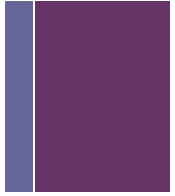


# + Exceptions

- An exception is a transfer of control to the OS kernel in response to some event (i.e., change in processor state)
  - Examples of events: Divide by 0, page fault, I/O request completes, typing Ctrl-C

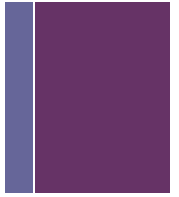


# + Asynchronous Exceptions (Interrupts)



- **Caused by events external to the processor**
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to “next” instruction
- **Examples:**
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Arrival of a packet from a network
    - Arrival of data from a disk

# + Synchronous Exceptions

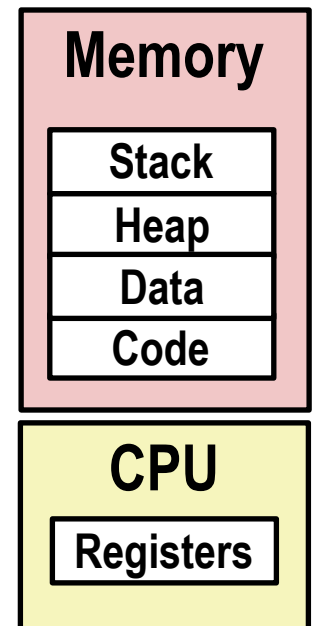


- **Caused by events that occur as a result of executing an instruction:**
  - Traps
    - Intentional
    - Example: *system calls*
    - Returns control to “next” instruction
  - Faults
    - Unintentional but possibly recoverable
    - Example: *page fault*
    - Either re-executes faulting (“current”) instruction or aborts
  - Aborts
    - Unintentional and unrecoverable
    - Example: *illegal memory access*
    - Aborts current program

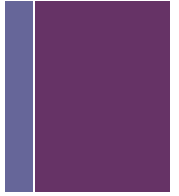
# + Processes



- A *process* is an instance of a running program.
  - One of the most successful ideas in computer science
  - Not the same as “program”
- Process provided with two key abstractions by OS:
  - *Logical control flow*
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called *context switching*
  - *Private address space*
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called *virtual memory*



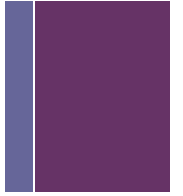
# + Creating and Terminating Processes



- From a programmer's perspective, we can think of a process as being in one of three states
- **Running**
  - Process is either executing, or waiting to be executed and will eventually be scheduled (i.e., chosen to execute) by the kernel
- **Stopped**
  - Process execution is suspended and will not be scheduled until further notice
- **Terminated**
  - Process is stopped permanently

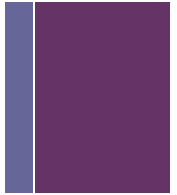


# + Process Management



- **Creating processes**
  - Call `fork`
  - One call, *two* returns
- **Process completion**
  - Call `exit`
  - One call, no return
- **Reaping and waiting for processes**
  - Call `wait` or `waitpid`
- **Loading and running programs**
  - Call `execve`
  - One call, no return

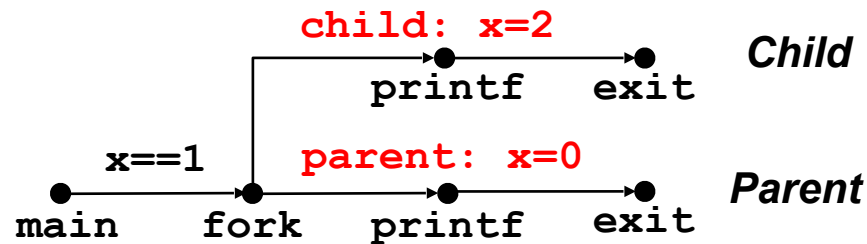
# + Process Graph Example



```
int main() {
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) { /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```



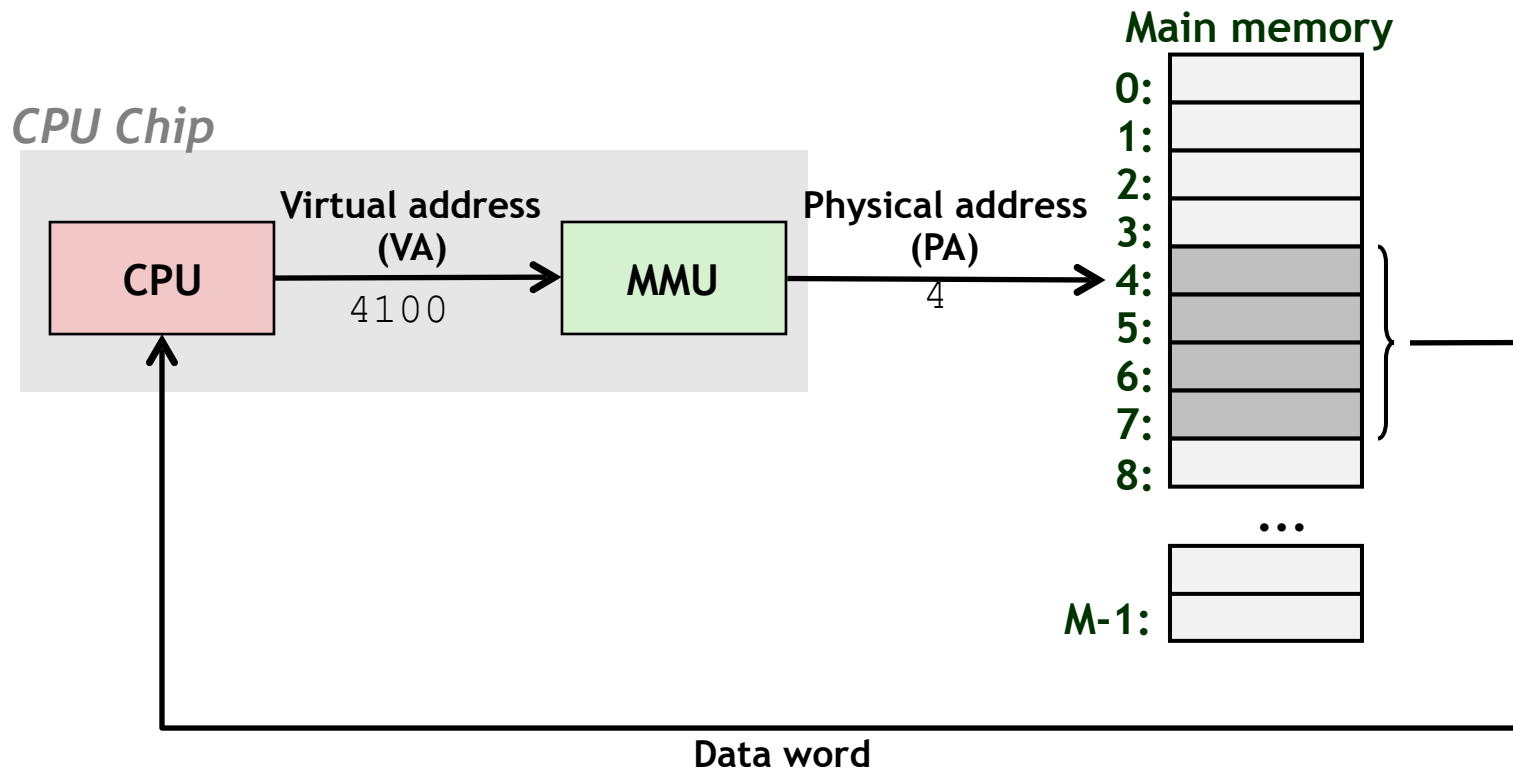


+

Virtual Memory

# + Virtual Addressing

- Creates illusion to process that it has total address space.
- Used in all modern, non-trivial systems

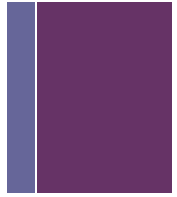


# + Why Virtual Memory (VM)?

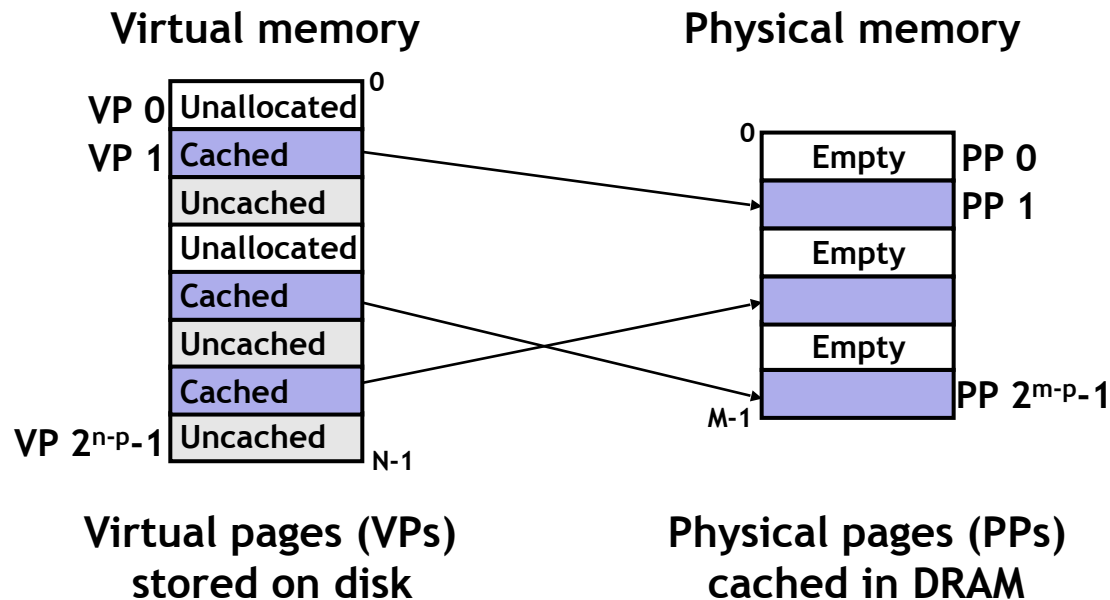


- **Uses main memory efficiently**
  - Use DRAM as a cache for parts of a virtual address space
- **Simplifies memory management**
  - Each process gets the same uniform linear address space
- **Isolates address spaces**
  - One process can't interfere with another's memory
  - User program cannot access privileged kernel information and code

# + VM as Caching

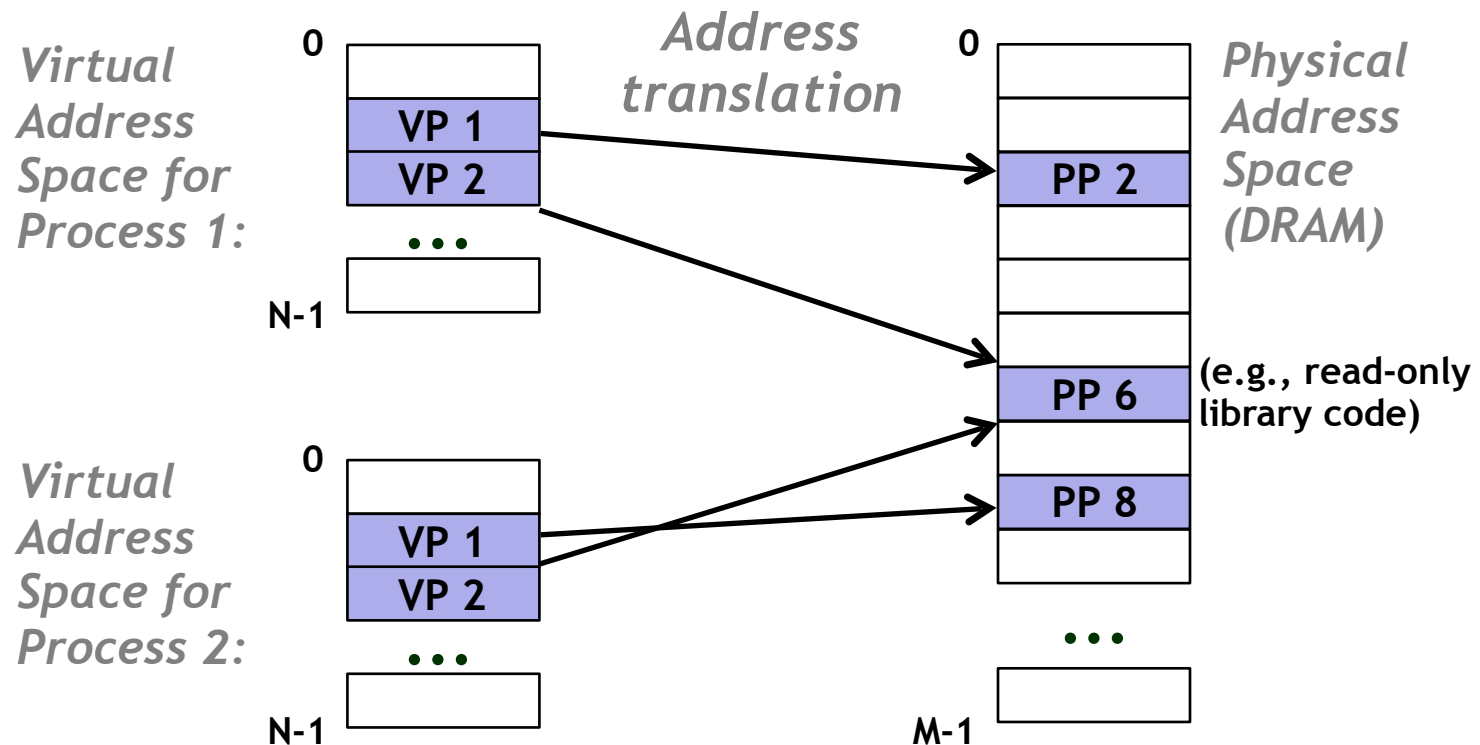


- Conceptually, virtual memory is an array of  $N$  contiguous bytes stored on disk.
- The contents of the array on disk are cached in physical memory
  - These cache blocks are called *pages*



# + VM as a Tool for Memory Management

- **Key idea: each process has its own virtual address space**
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory

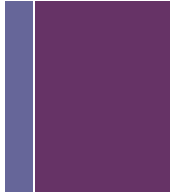




Dynamic Allocation



# + Constraints



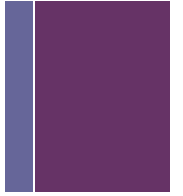
- **Applications**

- Can issue arbitrary sequence of `malloc` and `free` requests
- `free` request must be to a `malloc`'d block

- **Allocators**

- Can't control number or size of allocated blocks
- Can't reorder or buffer requests for memory
- Must allocate blocks from *free* memory
- Can manipulate and modify only *free* memory
- Can't move the allocated blocks once they are `malloc`'d

# + Performance Goals

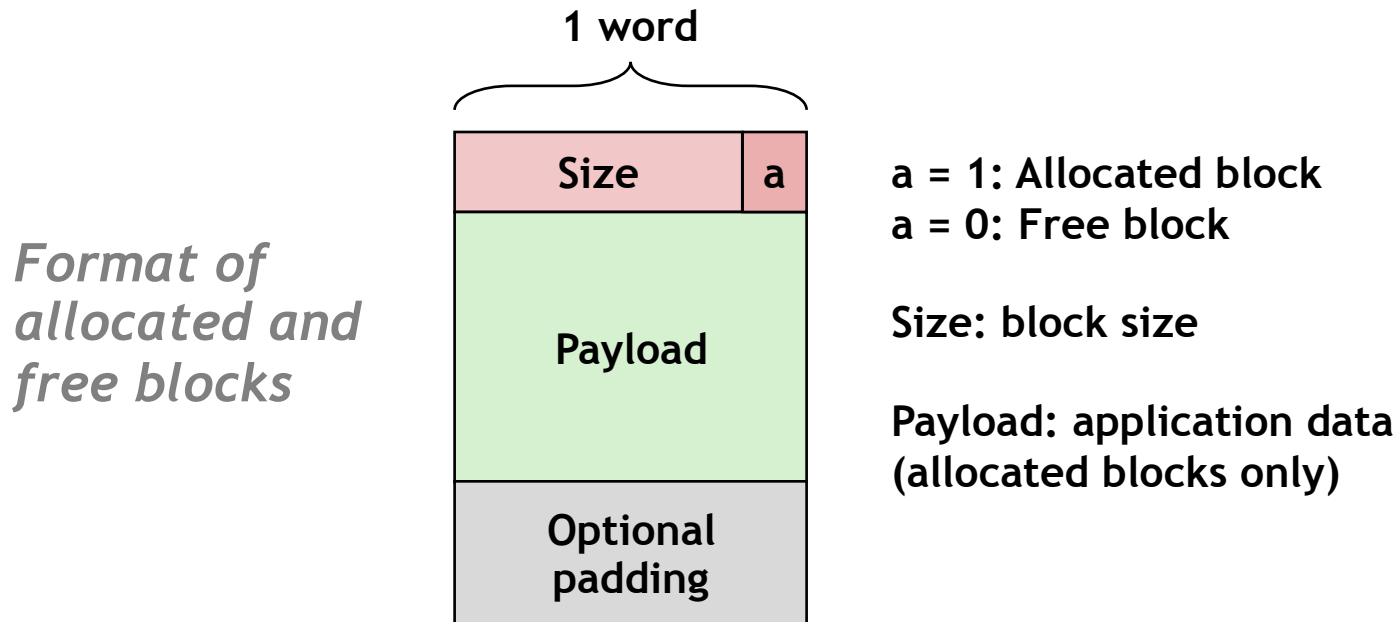


- **Goal: Increase utilization (by reducing fragmentation)**
  - internal fragmentation
  - external fragmentation
- **Goal: Maximize throughput:**
  - Number of completed requests per unit time
  - Example:
    - 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
    - Throughput is 1,000 operations/second

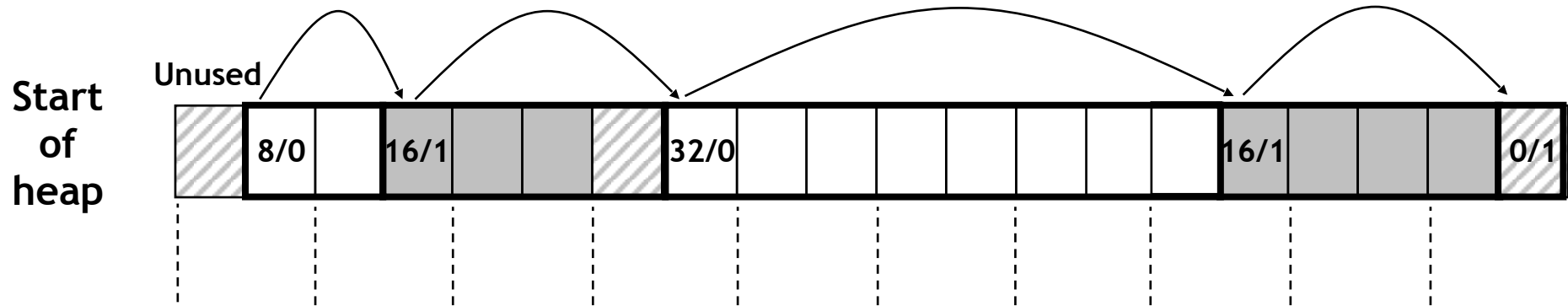
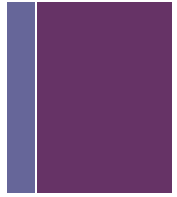
# + Implicit Lists



- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are word-aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - *(Note: when reading size, must mask out any extra bits)*



# + Detailed Implicit Free List Example



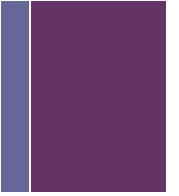
8-byte word  
aligned

Allocated blocks: shaded

Free blocks: unshaded

Headers: labeled with size in  
bytes/allocated bit

# + Ideas to remember...



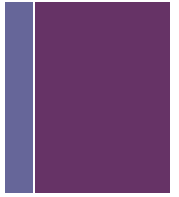
- **Allocation**
- **Freeing**
- **Fragmentation**
  - External
  - Internal
- **Coalescing**



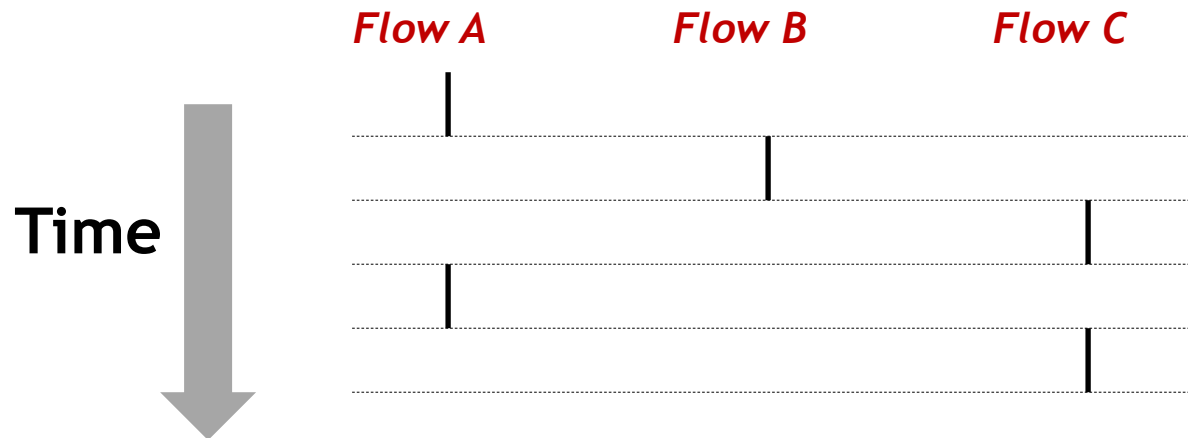
+

Concurrent Programming

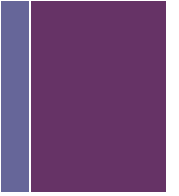
# + Concurrency



- Multiple logical control flows.
- Flows run concurrently if they overlap in time
  - Otherwise, they are sequential
- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



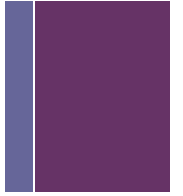
# + Process Concurrency



- Use fork to launch multiple processes to do work.
  - Hard for them to share data. Must use IPC.
- How to communicate across processes? (*inter-process communication or IPC*)
  - via *sockets*
  - via *pipes*
  - via *shared memory objects*



# + Thread-based Concurrency



- **A thread is...**
  - a unit of execution, associated with a process.
  - the smallest sequence of instructions that can be managed independently by the OS scheduler
- **Multiple threads can..**
  - exist within one process
  - be executing concurrently
  - *share resources* such as memory

# + Threads vs. Processes



- **Similarities**

- Each has its own logical control flow
- Each can run concurrently (possibly on different cores)
- Each is context switched

- **Differences**

- Threads share all code and data (except local stacks)
  - Processes do not
- Threads are somewhat less expensive than processes
  - Process control (creating/reaping) **2x** as expensive as thread control

# + Concurrent Programming is Hard!



- The ease with which threads share data and resources also makes them vulnerable to subtle and baffling errors.
- Classical problem classes of concurrent programs:
  - *Races*: outcome depends on arbitrary scheduling decisions elsewhere in the system
  - *Deadlock*: improper resource allocation prevents forward progress

# + Race Example

- What's the expected output on line 11?
  - 2
- Possible output...
  - 1

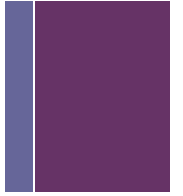
```
1  int numbers[2] = { 1, 1 };
2  int sum = 0;
3
4  int main() {
5      pthread_t tid;
6      pthread_create(&tid, NULL, run, numbers[1]);
7      for (int i = 0; i < 1; i++) {
8          sum += numbers[i];
9      }
10     pthread_join(tid, NULL);
11     printf("sum is %d\n", sum);
12 }
13
14 void* run(void* arg) {
15     int* numbers = (int*) arg;
16     for (int i = 0; i < 1; i++) {
17         sum += numbers[i];
18     }
19     return NULL;
20 }
```

# + Synchronizing Threads



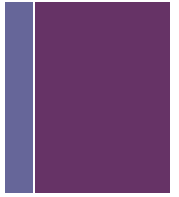
- **Shared variables are sometimes useful but they introduce the possibility of *synchronization* errors.**
  - Like the one we saw last time
- **How do we prevent such things?**
  - We need to make sure that only one thread is mutating shared variables at a time.
  - This is known as *mutual exclusion*
- **Moreover, we must protect *critical sections*.**

# + Mutual Exclusion



- A *mutex*...
  - is synchronization variable that is used to protect the access to shared variables.
  - surrounds critical sections so that one threads is allowed inside at a time.
- **However we have to be careful with these because they can lead to code that performs poorly, or in some cases not at all...**

# + Deadlock



- Both processes need resources to continue execution.
- P1 requires additional resource R1 and is in possession of resource R2
- P2 requires additional resource R2 and is in possession of R1; neither process can continue.

