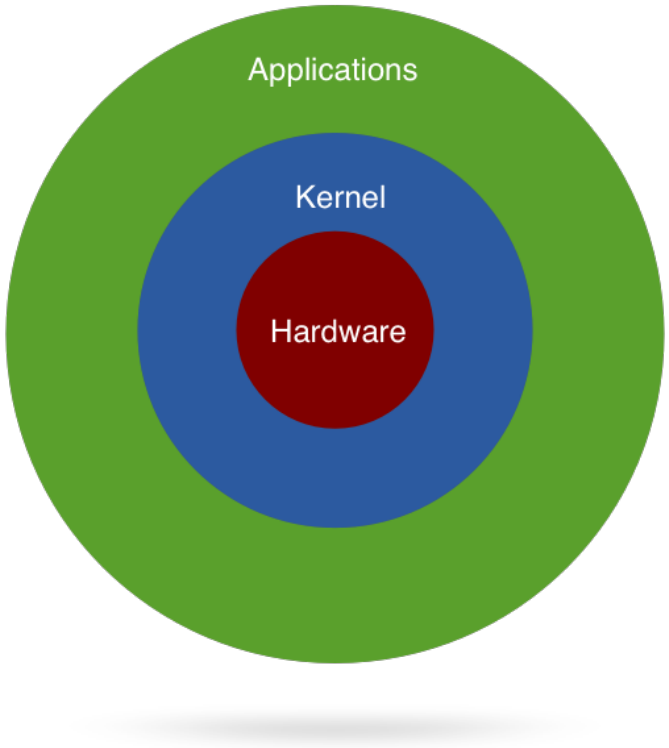
Exam 1 Review

Kernel



Kernel - The part of the OS that implements basic functionality and is always resident in memory.

The Operating System as an Extended Machine

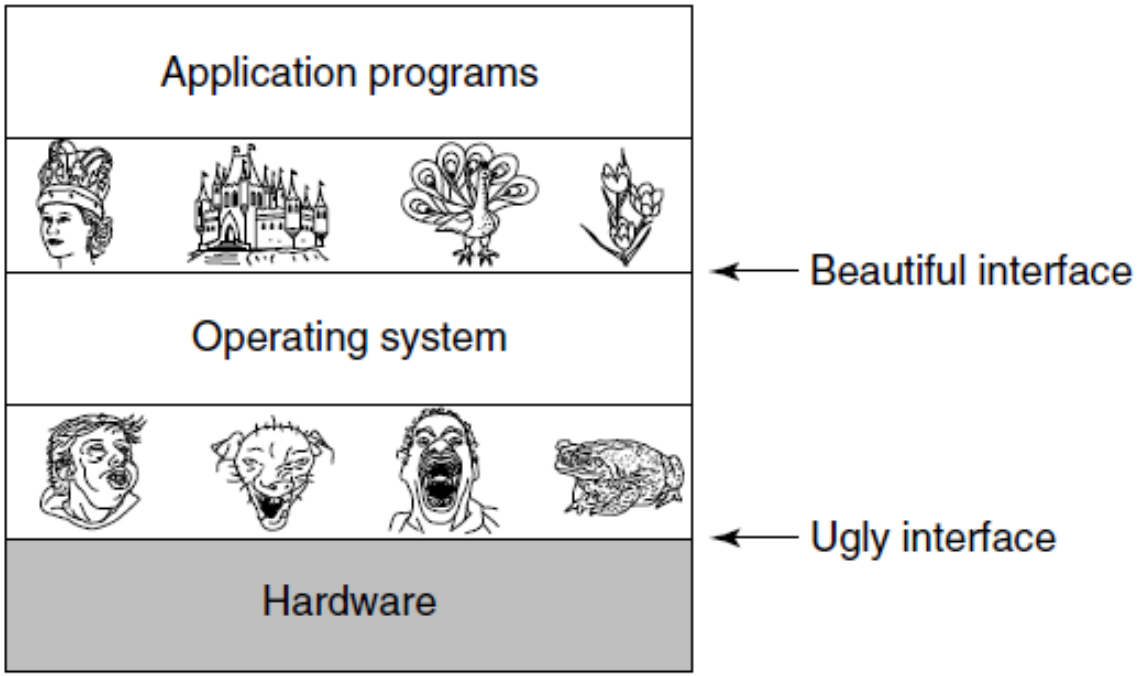


Figure 1-2. Operating systems turn ugly hardware into beautiful abstractions.

The Operating System's Role

- Provide the user with a cleaner model of the computer
  - An abstracted interface to the resources
- » Manage the resources

Interrupts

- Mechanism used by the OS to signal the system that a high-priority event has occurred that requires immediate attention.
  - I/O drives a lot of interrupts. Mouse movements, disk reads, etc
- The controller causing the interrupt places the interrupt number in an interrupt register. The OS must then take action

Interrupt Vector

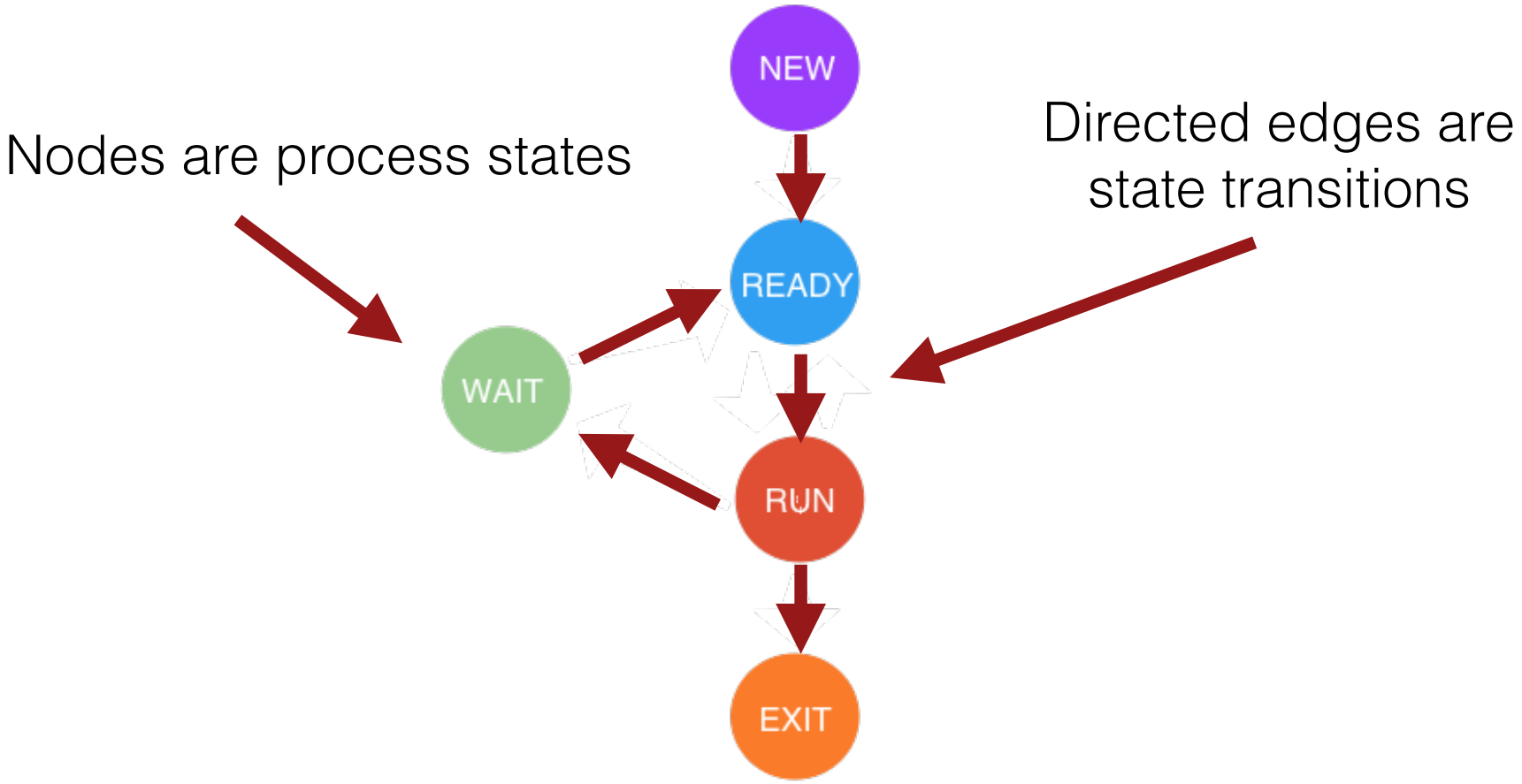
- Normal technique for handling interrupts is a data structure called the interrupt vector.
- One entry for each interrupt.
- Each entry contains the address for the interrupt service routine.
- Some small hardware devices don't provide an interrupt system and instead uses an event loop. This is known as a status-driven system.

Programs v. Processes

- Program - a sequence of instructions written to perform a specified task.
- Process - an instance of a program in execution.
  - Process is associated with an address space
- A computer program is a passive collection of instructions; a process is the actual execution of those instructions.

© 2015 Trevor Bakker and The University of Texas at Arlington

Process State Diagram



Parent and children PIDs

- Why does fork return the child's PID to parent processes but it returns 0 to the children?
  - Provides a way to determine which process you are in.
  - Processes can only have one parent. To determine the parent PID you can call getppid().
  - Processes can have multiple children so there is no way for a function to give a parent its child PID

Process Creation

- After fork() both the parent and child continue executing with the instruction that follows the call to fork().
- The child process is a copy of the parent process. The child gets:
  - copy of the parent's data space
  - copy of the parent's heap
  - copy of the parent's stack
  - text segment if it's read-only

Copy-On-Write

- On Linux the parent process's pages are not copied for the child process.
- The pages are shared between the child and the parent process.
  - Pages are marked read-only
- When either process modifies a page, a page fault occurs and a separate copy of that particular page is made for that process which performed the modification.
- This process will then use the newly copied page rather than the shared one in all future references. The other process continues to use the original copy of the page

Inherited Properties

- user ID, group ID
- process group ID
- controlling terminal
- current working directory
- root directory
- file mode creation mask
- signal mask
- environment
- attached shared memory segments

Unique Properties

- the return value from fork()
- the process IDs
- the parent process IDs
- file locks
- pending alarms are cleared for the child
- the set of pending signals for the child is set to zero

Process Execution Modes

- Privileged - OS kernel processes which can execute all types of hardware operations and access all memory
- User Mode - Can not execute low-level I/O. Memory protection keeps these processes from trashing memory owned by the OS or other processes.

How does the OS track a process?

- Each process has a unique process identifier, or PID
  - The POSIX standard guarantees a PID as a signed integral datatype.
  - The datatype is an opaque type called pid_t

Process Control Block

- The kernel maintains a data structure to keep track of all the process information called the process control block or (PCB).
- The PCB also includes pointers to other data structures describing resources used by the process such as files (open files table) and memory (page tables).
- Maintains the state of the process
  - Over 170+ fields

Process Control Block

- Every task also needs its own stack
- So every task, in addition to having its own code and data, will also have a stack-area that is located in user-space, plus another stack-area that is located in kernel-space
- Each task also has a process-descriptor which is accessible only in kernel-space

Process Tables

- The OS holds the process control blocks in the process table
- Usually implemented as an array of pointers to process control block structures
  - Linux calls the PCB task_struct

Process Creation

- Processes are created when an existing process calls the fork() function
- New process created by fork is called the *child process*
- fork() is a function that is called once but returns twice
  - Only difference in the return value. Returns 0 in the child process and the child's PID in the parent process

fork() code example

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
{
  pid_t pid = fork();

  if (pid == -1) {
    // When fork() returns -1, an error happened.
    perror("fork failed");
    exit(EXIT_FAILURE);
  }
  else if (pid == 0) {
    // When fork() returns 0, we are in the child process.
    printf("Hello from the child process!\n");
    fflush(NULL);
    exit(EXIT_SUCCESS);
  }
  else {
    // When fork() returns a positive number, we are in the parent process
    // and the return value is the PID of the newly created child process.
    int status;
    (void)waitpid(pid, &status, 0);
    printf("Hello form the parent process!");
    fflush(NULL);
  }
  return EXIT_SUCCESS;
}
```

I/O is buffered. Make sure to use `flush()`

waitpid() code example

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
  pid_t child_pid = fork();
  int status;

  if (child_pid == 0)
  {
    // Sleep for a second
    sleep(1);

    // Intentionally SEGFAULT the child process
    int *p = NULL;
    *p = 1;

    exit(0);
  }

  // Wait for the child to exit
  waitpid( child_pid, &status, 0 );

  // See if the child was terminated by a signal
  if( WIFSIGNALED( status ) )
  {
    // Print the signal that the child terminated with
    printf("Child returned with status %d\n", WTERMSIG( status ) );
  }

  return 0;
}
```

`exec` functions

- When exec is called the new program, specified by exec, completely replaces the running process.
  - text, data, heap and stack are all replaced
  - PID stays the same since it's not a new process

exec code example

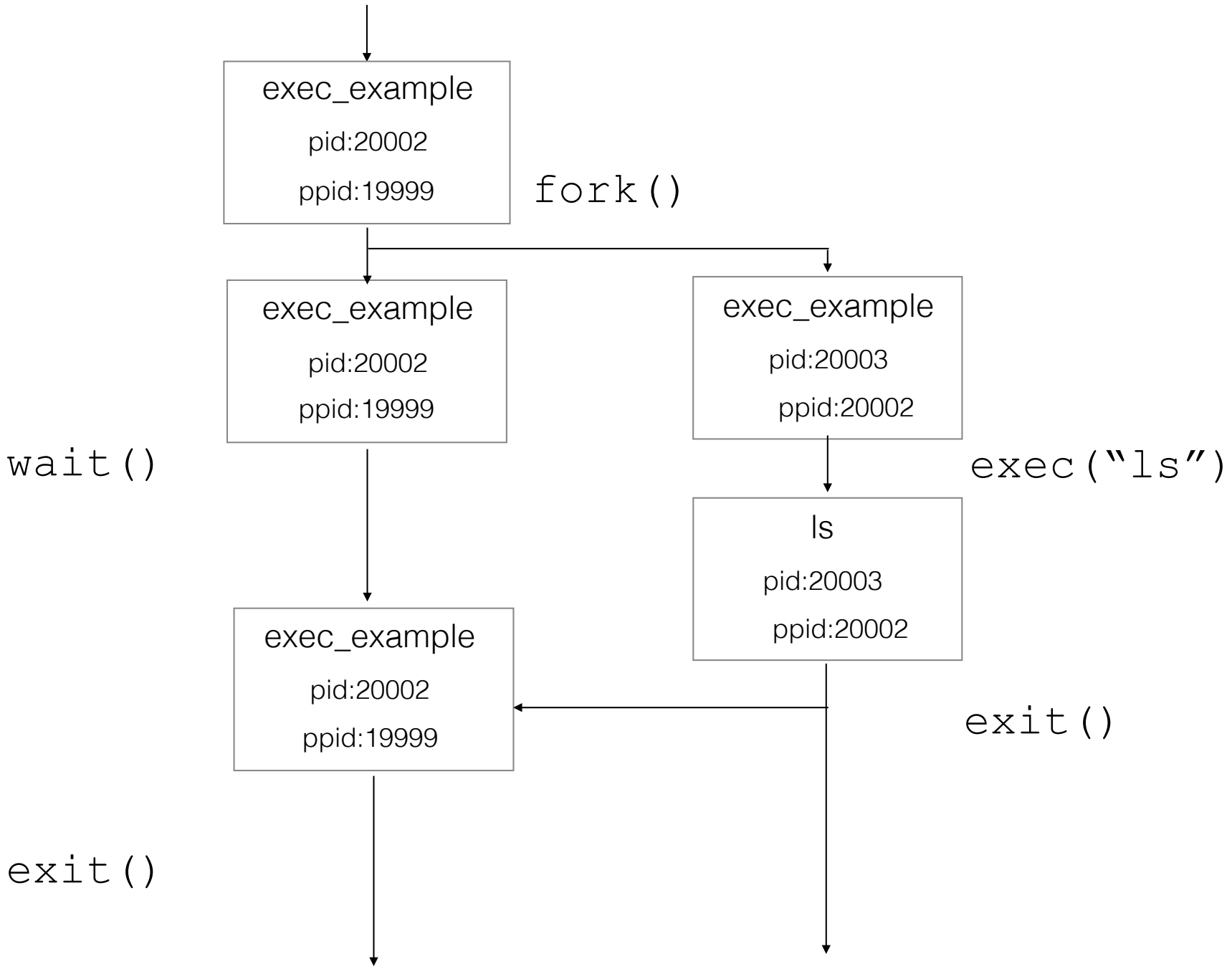
```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

int main(void)
{
  pid_t child_pid = fork();
  int status;

  if (child_pid == 0)
  {
    execl("/bin/ls", "ls", NULL );
    exit(0);
  }

  // Wait for the child to exit
  waitpid( child_pid, &status, 0 );

  return 0;
}
```



Kernel Mode v. User Mode

- A process is executing either in user mode, or in kernel mode. Depending on which privileges, address space a process is executing in, we say that it is either in user space, or kernel space.

- When executing in user mode, a process has normal privileges and can and can't do certain things. When executing in kernel mode, a process has every privilege, and can do anything.

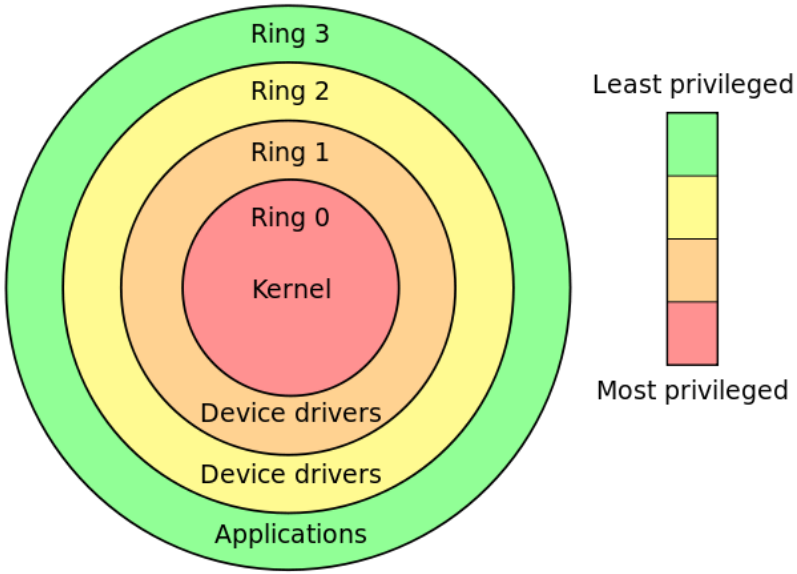
- Processes switch between user space and kernel space using system calls.

Kernel Mode v. User Mode

- These two modes aren't just labels; they're enforced by the CPU hardware.
- If code executing in User mode attempts to do something outside its purview such as accessing a privileged CPU instruction or modifying memory that it has no access to:
  - Trappable exception is thrown. Instead of your entire system crashing, only that particular application crashes.

x86 Protection Rings

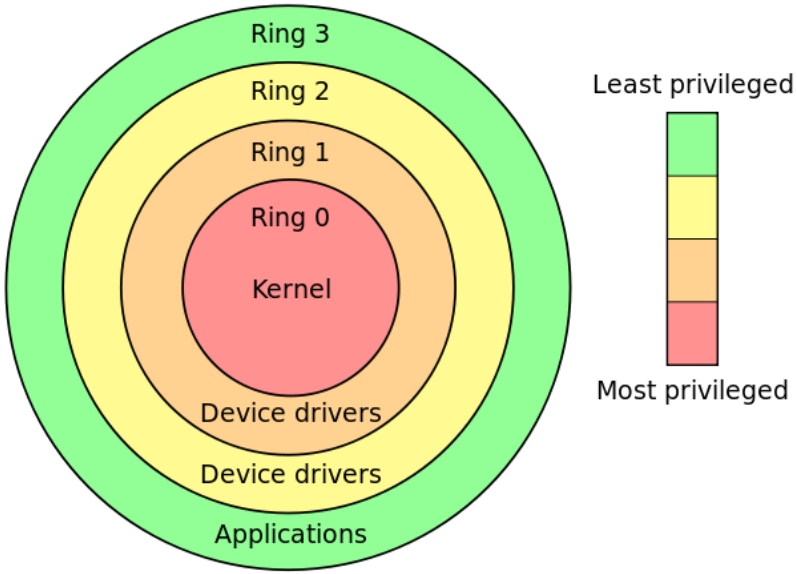
- Four privilege levels or rings, numbered from 0 to 3, with ring 0 being the most privileged and 3 being the least.
- Rings 1 and 2 weren't used in practice.
  - » VM's changed that



© 2015 Trevor Bakker and The University of Texas at Arlington

x86 Protection Rings

- Programs that run in Ring 0 can do anything with the system.
- Code that runs in Ring 3 should be able to fail at any time without impact to the rest of the computer system.



CPU Rings and Privilege

- CPU privilege level has nothing to do with operating system users.
- Whether you're root, Administrator, guest, or a regular user, it does not matter.
- All user code runs in ring 3 and all kernel code runs in ring 0, regardless of the OS user on whose behalf the code operates.

CPU Rings and Privilege

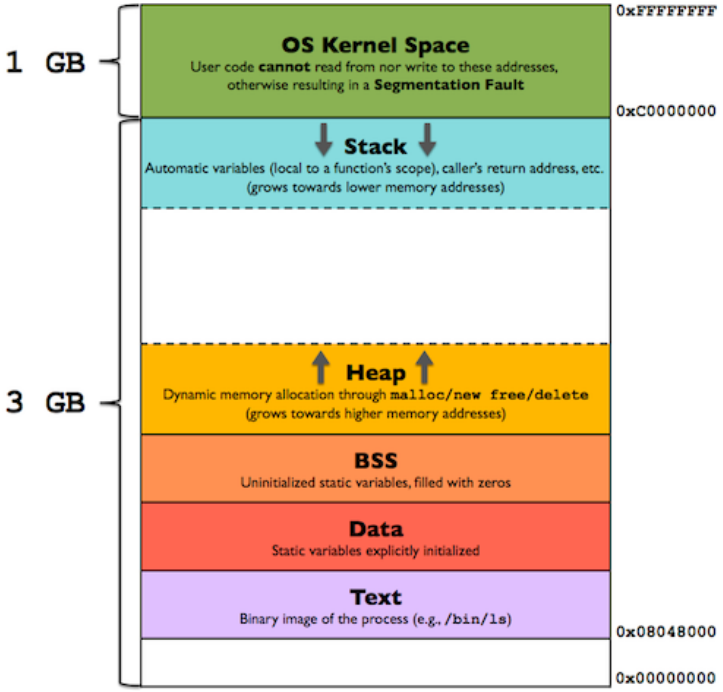
- Due to restricted access to memory and I/O ports, user mode can do almost nothing to the outside world without calling on the kernel.
  - It can't open files, send network packets, print to the screen, or allocate memory.
  - User processes run in a severely limited sandbox set up by ring zero.

© 2015 Trevor Bakker and The University of Texas at Arlington

Address Spaces

- An address space is the set of addresses in RAM that a process can use.
- Multiple programs in memory need OS to partition the available memory
  - Keep programs from interfering with each other and OS.

Address Spaces



32-bit address = 2^{32} = 4GB of address space

- Kernel gets upper 1 GB

System Calls

- How do our programs utilize the resources controlled by the OS or communicate with other process?

- Because user mode software can not access hardware devices directly, they must notify the operating system in order to complete system tasks. This includes displaying text, obtaining input from user, printing a document, etc.

System Calls

- Instead of directly calling a section of code the system call instruction issues an interrupt.
- By not allowing the application to execute code freely the operating system can verify that the application has appropriate privileges to call the function.
  - Only system calls enter the kernel. Procedure calls do not.

Monolithic Systems (1)

Basic structure of OS

1. A main program that invokes the requested service procedure.

2. A set of service procedures that carry out the system calls.

3. A set of utility procedures that help the service procedures.

Microkernel

- Only basic functionality is included in the kernel
  - What is basic? Only code that must run in supervisor mode because it must use privileged resources such as protected instructions
- Everything else runs in user space.

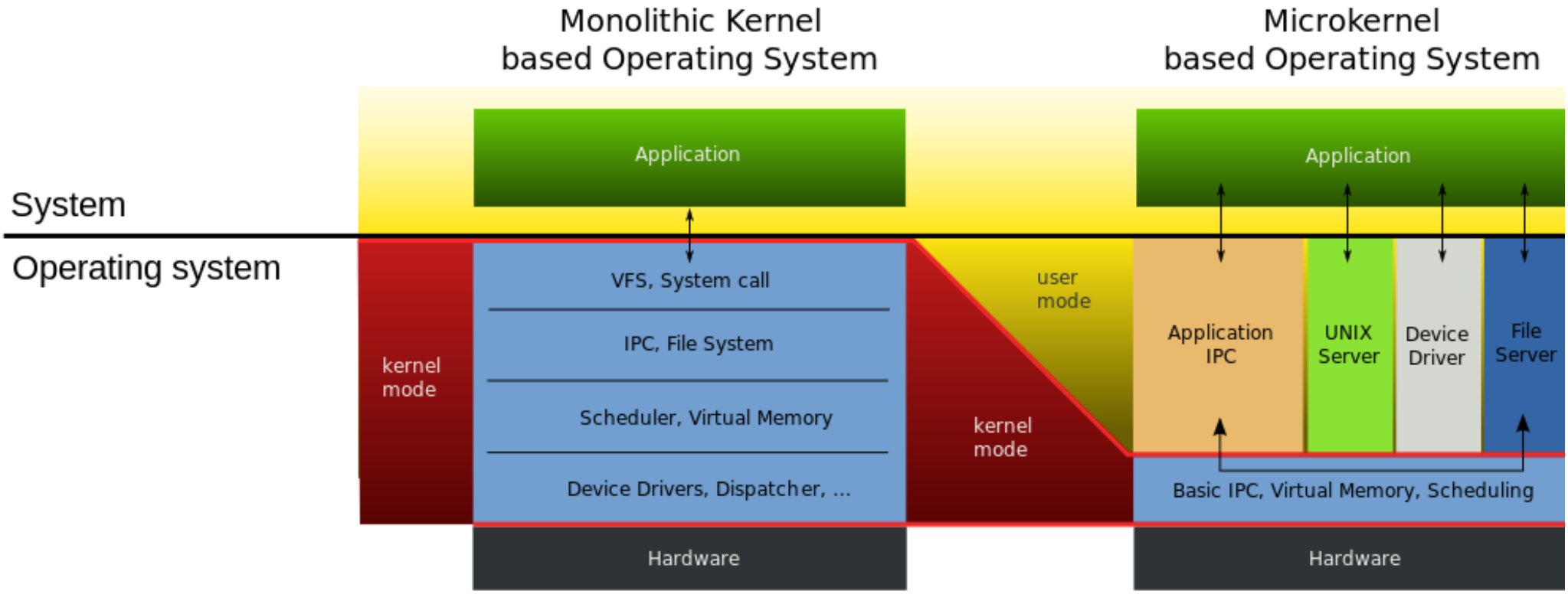
Microkernel

- Put the mechanism in the kernel and the policy in user-mode
  - For example: job scheduling
    - Kernel runs the highest priority process
    - User-mode job scheduler decides priorities

Microkernel

- Theoretically more robust since limiting the amount of code that runs in protected mode limits the number of catastrophic crashes
- Easier to inspect for flaws since a smaller portion of code exists
- May run slower since there are more interrupts from user space to kernel
- Example: Minix

Micro v. Monolithic



"OS-structure2" by Golftheman - http://en.wikipedia.org/wiki/Image:OS-structure.svg. Licensed under Public domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:OS-structure2.svg#mediaviewer/File:OS-structure2.svg

Part 2 goes here

Race Conditions

- Processes working together may share common storage.
- A race condition is where two or more processes are reading or writing shared data and the final result depends on who runs precisely when.
- Very common as multiprogramming with more and more cores

Critical Regions

- Key is to prevent more than one process form reading and writing from a shared area at the same time
  - Mutual exclusion
  - pthread mutexes

Critical Regions

» Critical region is the part of a program where shared memory is accesses.

» Four conditions to avoid race conditions

1. No two processes may be simultaneously inside the critical region
2. No assumptions may be made about speeds or number of CPUs
3. No process running outside its critical region may block any process
4. No process should have to wait forever to enter its critical region

Priority Inversion and Starvation

- Indefinite blocking or starvation
  - process is not deadlocked
  - but is never removed from the semaphore queue

- Priority inversion
  - lower-priority process holds a lock needed by higher-priority process !

- Assume three processes L, M, and H
  - Priorities in the order L < M < H
  - L holds shared resource R, needed by H
  - M preempts L, H needs to wait for both L and M !!

Priority Inversion and Starvation

- Solutions
  - Only support at most two priorities
  - Priority inheritance protocol – lower priority process accessing shared resource inherits higher priority

Scheduling

- Multiple processes or threads competing for the CPU
  - Choice needs to be made of which to run next
    - Scheduler - part of the OS that makes that decision
    - Scheduling algorithm

Scheduling

- I/O interrupt
  - Hardware clock provides 50 or 60 hz interrupts
- Scheduling algorithms can be divided into 2 categories based on how they deal with clock interrupts
  - Non-preemptive (cooperative)
  - Preemptive

Scheduling Algorithm Goals

- Batch Systems
  - Throughput - maximize jobs per hour
  - Turnaround time - minimize time between submission and termination
  - CPU utilization - you paid for that screaming CPU, use it

Scheduling Algorithm Goals

- Interactive systems
  - Response time
  - Proportionality - meets users expected performance

Scheduling Algorithm Goals

- Real-time systems
  - Meet deadlines
  - Predictability - avoid quality degradation in multimedia systems

Scheduling Metrics

- Throughput - number jobs per unit of time
- Turnaround time - average of time when jobs are submitted to when they complete
- Response time - average of time when jobs are submitted to jobs start running
- Wait time - time jobs spend in the wait queue

FCFS Scheduling

- First come, first served algorithm (FCFS).
- Easy to implement
- Well understood by anyone
- The fairest algorithm. No process is favored over another.

FCFS

Process ID	Arrival Time	Runtime
1	0	20
2	2	2
3	2	2

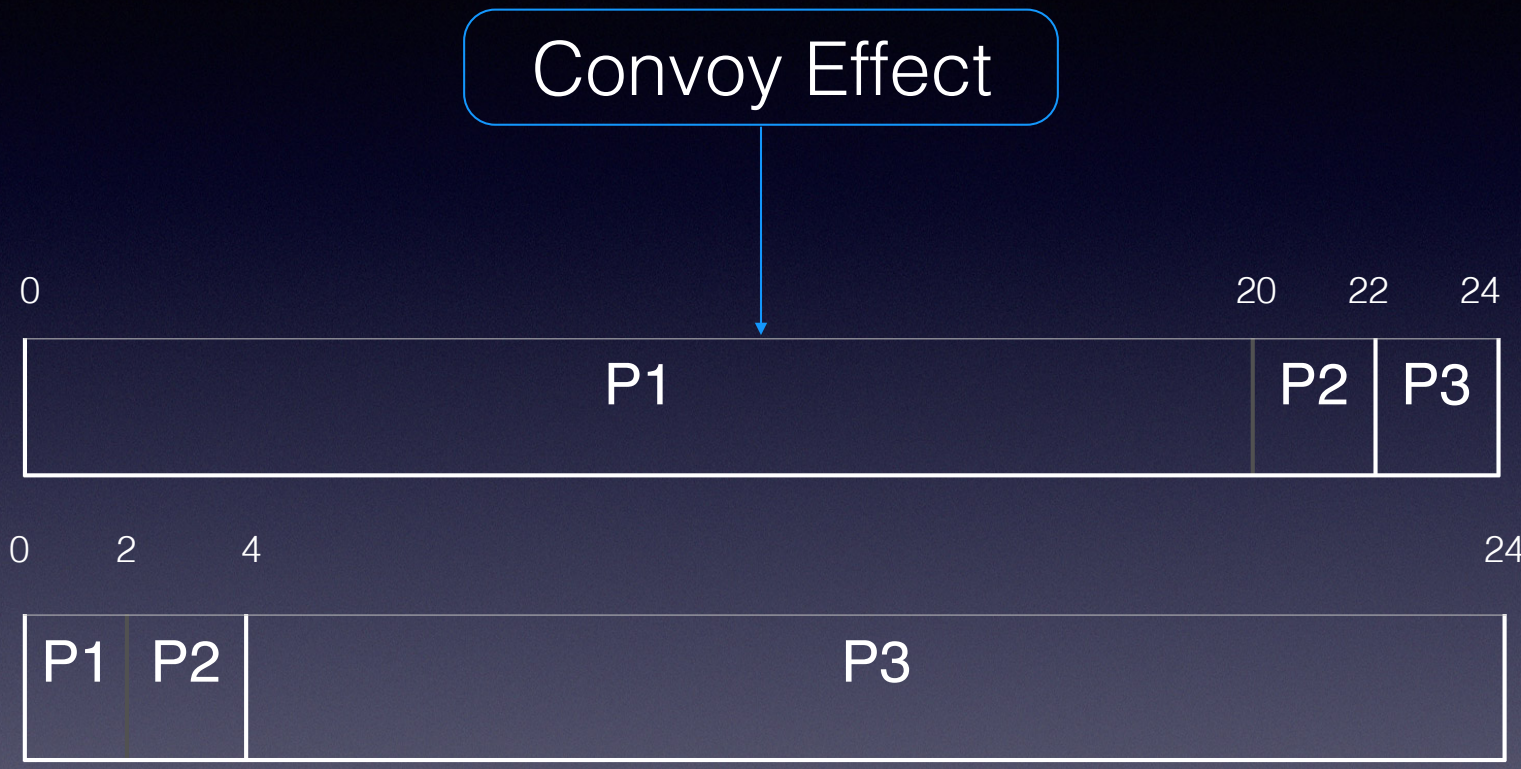
Average Waiting Time: (0 + 18 + 20) / 3 = 12.67

FCFS

Process ID	Arrival Time	Runtime
1	0	2
2	2	2
3	2	20

Average Waiting Time: (0 + 0 + 2) / 3 = 0.67

FCFS



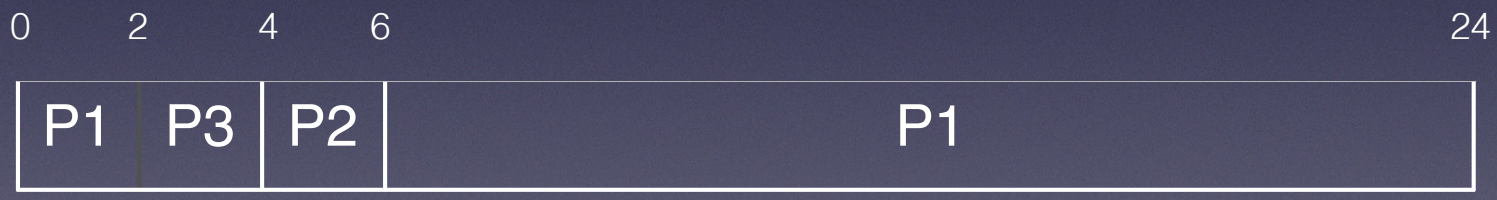
Very Dependent On Job Arrival Time

Preemption

Without Preemption	With Preemption
FCFS	Round Robin
SRTF	Shortest Remaining Time Next
Priority	Priority With Preemption

FCFS w/ Priority (Round Robin)

Process ID	Arrival Time	Runtime	Priority
1	0	20	4
2	2	2	2
3	2	2	1



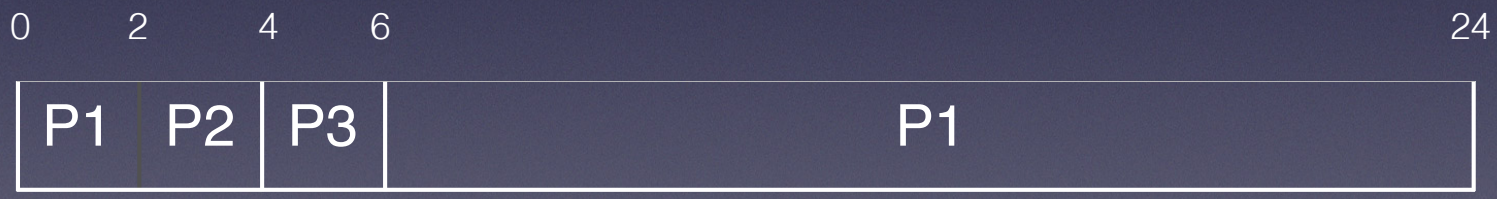
Average Waiting Time: (4 + 2 + 0) / 3 = 2

Time Quantum

- Choosing length is a problem
  - Context switching is expensive
  - Still need responsiveness

SJN with Preemption

Process ID	Arrival Time	Runtime
1	0	20
2	2	2
3	2	2



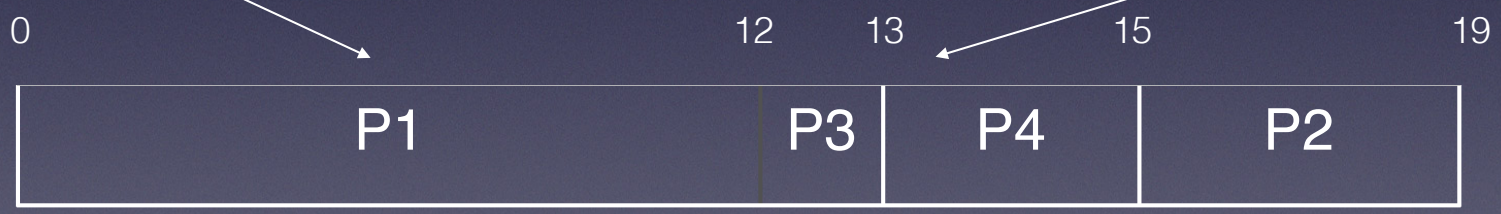
Average Waiting Time: (4 + 0 + 2) / 3 = 2

SJN

Process ID	Arrival Time	Runtime
1	0	12
2	2	4
3	3	1
4	4	2

P1 runs first since it's the only process

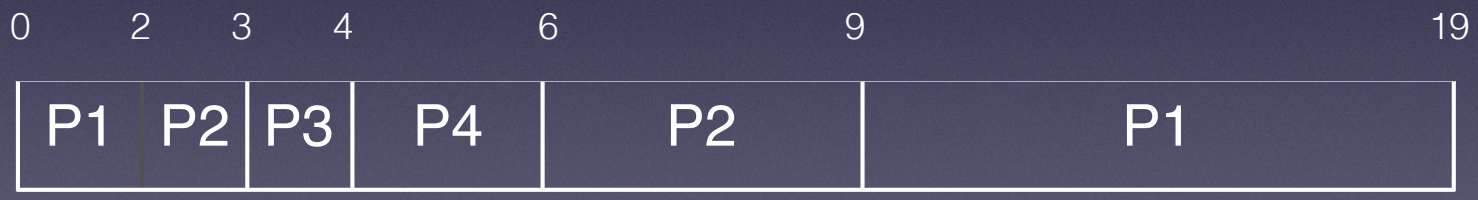
No preemption so P3, P4, and P2 wait. Then sorted by least runtime



Average Waiting Time: (0+13+9+9)/4 = 7.75

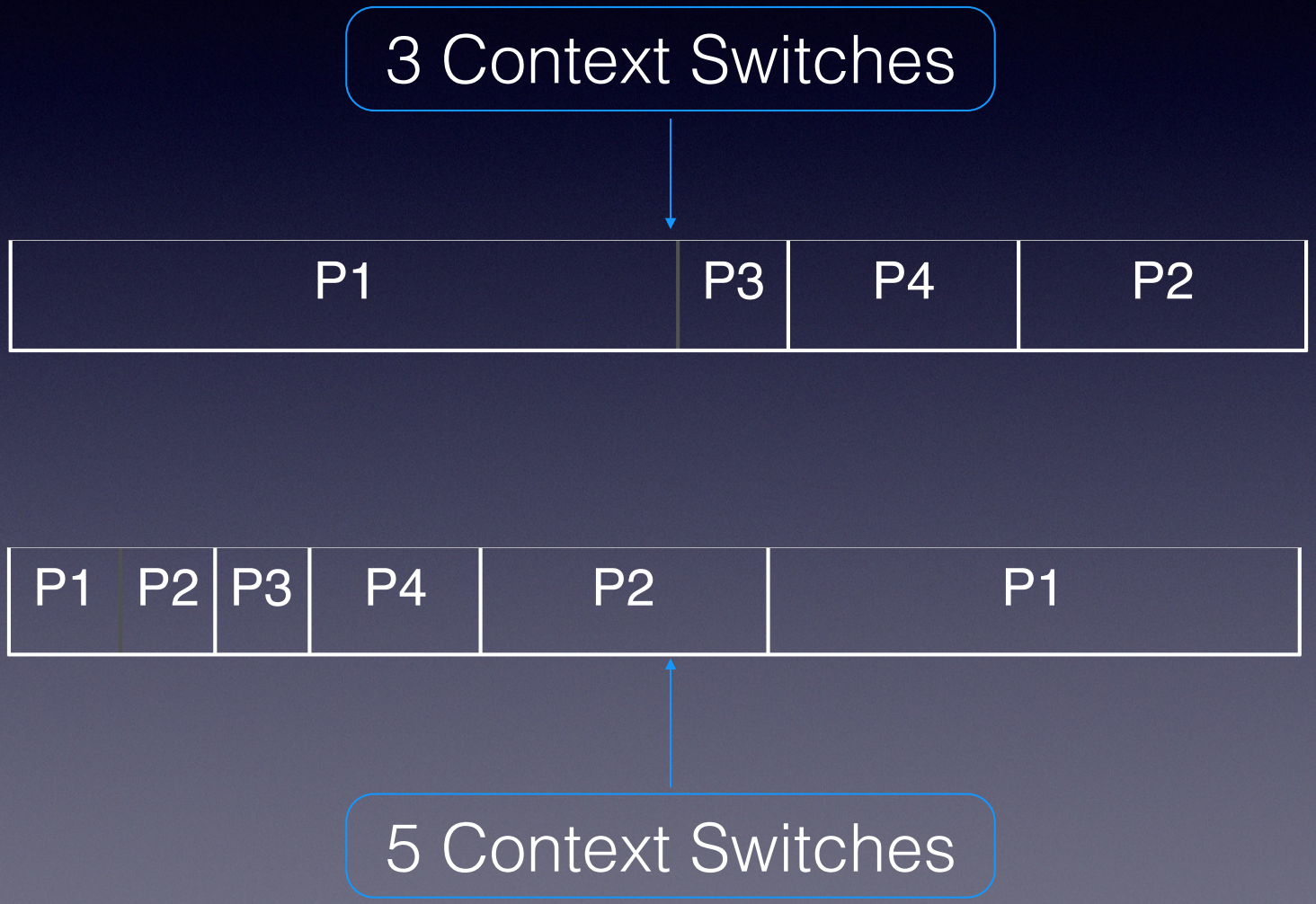
SJN With Preemption

Process ID	Arrival Time	Runtime
1	0	12
2	2	4
3	3	1
4	4	2



Average Waiting Time: (7+3+0+0)/4 = 2.5

SRTF Without and With Preemption



Lottery Scheduling

- Process receives a lottery ticket
- Lottery ticket chosen at random and the winning process is allowed to run.
- More important processes can be given more lottery tickets.
- Highly responsive.
- Tickets can be exchanged by cooperating processes

Lottery Scheduling

- Lottery scheduling can solve problems difficult to handle by other schedulers.
- Video server with video streams of different frame rates. (10, 20, 25 frames per second)
  - Processes get 10, 20, 25 tickets

Binding

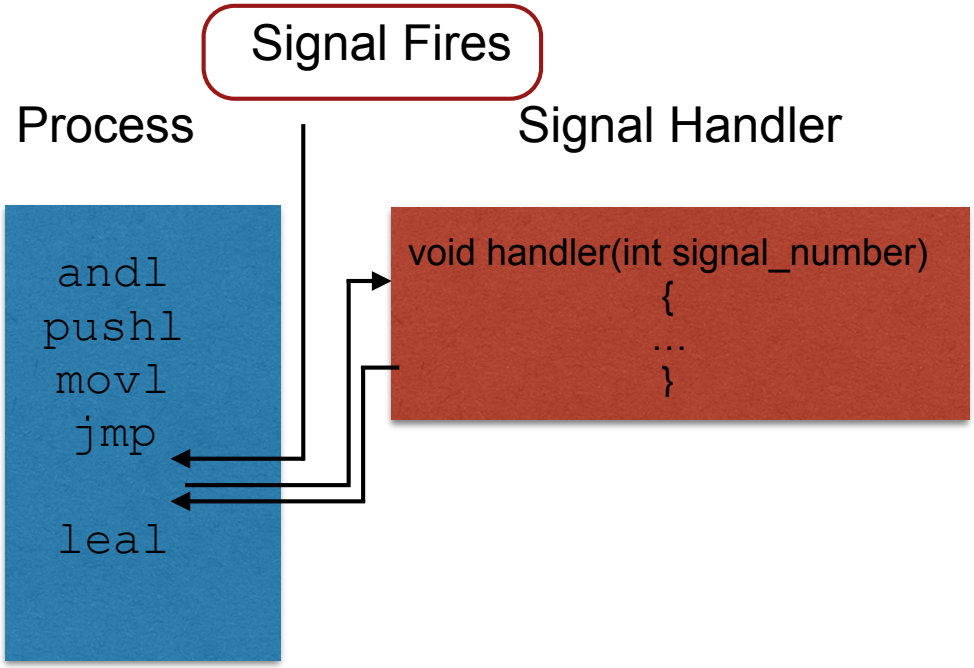
The process of determining where in the physical memory the subroutine should go and making the reference in the main routine point to the subroutine.

Binding Model

- Coding the program
- Translating into object module
  - Compiler, Assembler, Interpreter
- Linking with other modules
- Loading into primary memory
- Running the process

Signals

- A signal is a asynchronous notification of an event
- Signals are how the operating system communicates with an application process.
- When a signal is received the process stops running and the assigned signal handler is run.
- When the signal has been handled the process resumes where it left off.



Signal Terminology

- A signal is *generated* for a process when the event that causes the signal occurs. The event can be a hardware exception, a software condition, or a call to the kill function
- When a signal is generated the kernel usually sets a flag in the process table

Signal Terminology

- A signal is *delivered* to a process when the action for a signal is taken.
- During the time between the generation of the signal and the delivery of the signal the signal is said to be *pending*.

Signal Handling

- Every signal is assigned a default handler. Usually they just exit the process.
- Programs can install their own signal handlers for most signals.
  - Can’t install handlers for:
    - SIGKILL
    - SIGSTOP

Memory Hierarchy

- How does the operating system create abstractions from memory, and how does it manage them?
- Memory hierarchy:
  - Few megabytes of very fast, very expensive, volatile cache memory
  - A few gigabytes of medium-speed, medium priced, volatile main memory
  - A few terabytes of slow, cheap, nonvolatile magnetic or solid state disk storage.

Memory Hierarchy

- Memory Manager manages it
  - Tracks which parts of memory are in use
  - Allocates memory to processes
  - Deallocates memory when processes are done.

No Memory Abstraction

- Early mainframes (before 1960), early minicomputers (before 1970), early personal computers (before 1980) had no memory abstraction
- Every program saw physical memory.
- Programmers saw a set of addresses from 0 to some maximum.

No Memory Abstraction

- Two programs could not be resident in memory at the same time.
  - Address conflicts
  - Could run multi-threaded since threads share address space.
  - Limited use. Users want unrelated programs to be running at once.

No Memory Abstraction

- Multiple programs possible
  - Save entire contents of memory to a disk file
  - Bring in new and run it
  - Swapping.
- Additional hardware will allow concurrency without swapping.

Address Spaces

- Exposing memory to processes have several drawbacks
  - If user programs can address every byte of memory, the OS can be trashed intentionally or by accident.
  - Difficult to have multiple programs running at once

Address Spaces

- Abstract memory
  - Process creates a virtual CPU abstraction
  - Address space is abstract memory for a process to live in
- Address space: Set of all addresses that a process can see to address memory
- Each process has an independent address space

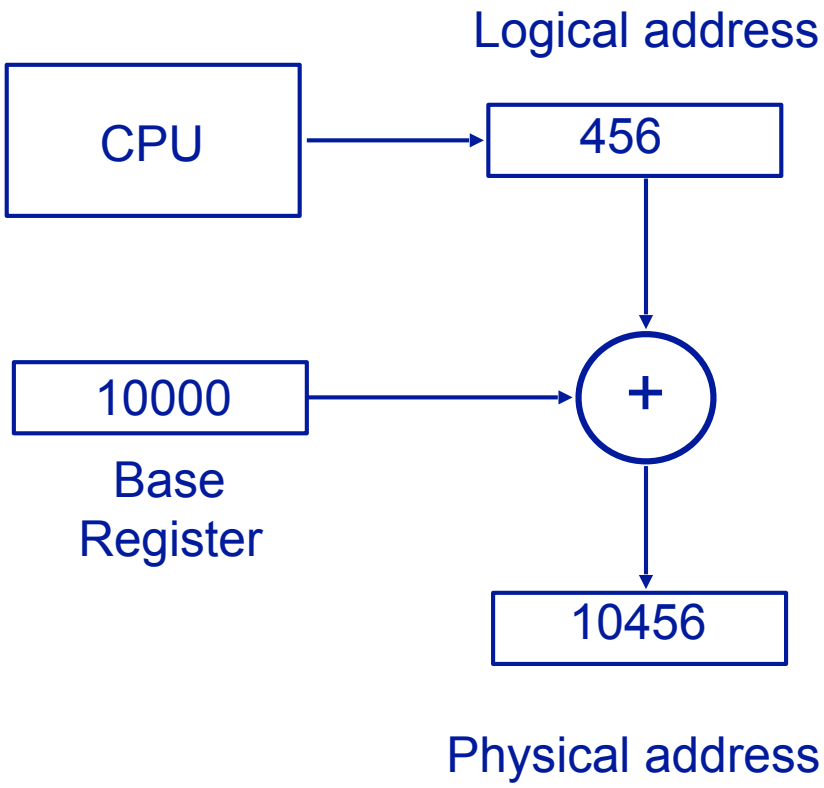
Address Spaces

- Need to map address 28 in one process and address 28 in another process to a separate physical address.
- Dynamic relocation
- Older solution: two special registers
  - Base register
  - Limit register

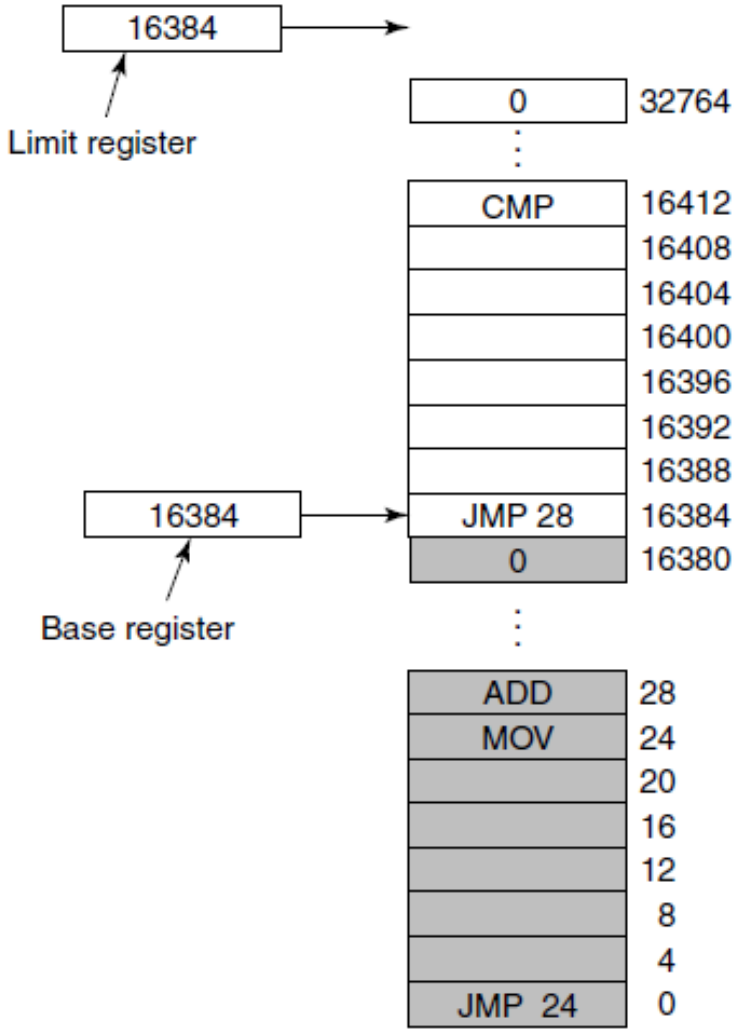
Physical v. Logical Addresses

- Originally programs were compiled to reference addresses that corresponded one to one with physical memory addresses.
- Unknowingly using concept of logical and physical memory
  - Logical addresses - Set of addresses the CPU generates as the program is executed.
  - Physical addresses - Set of addresses used to reference physical memory.

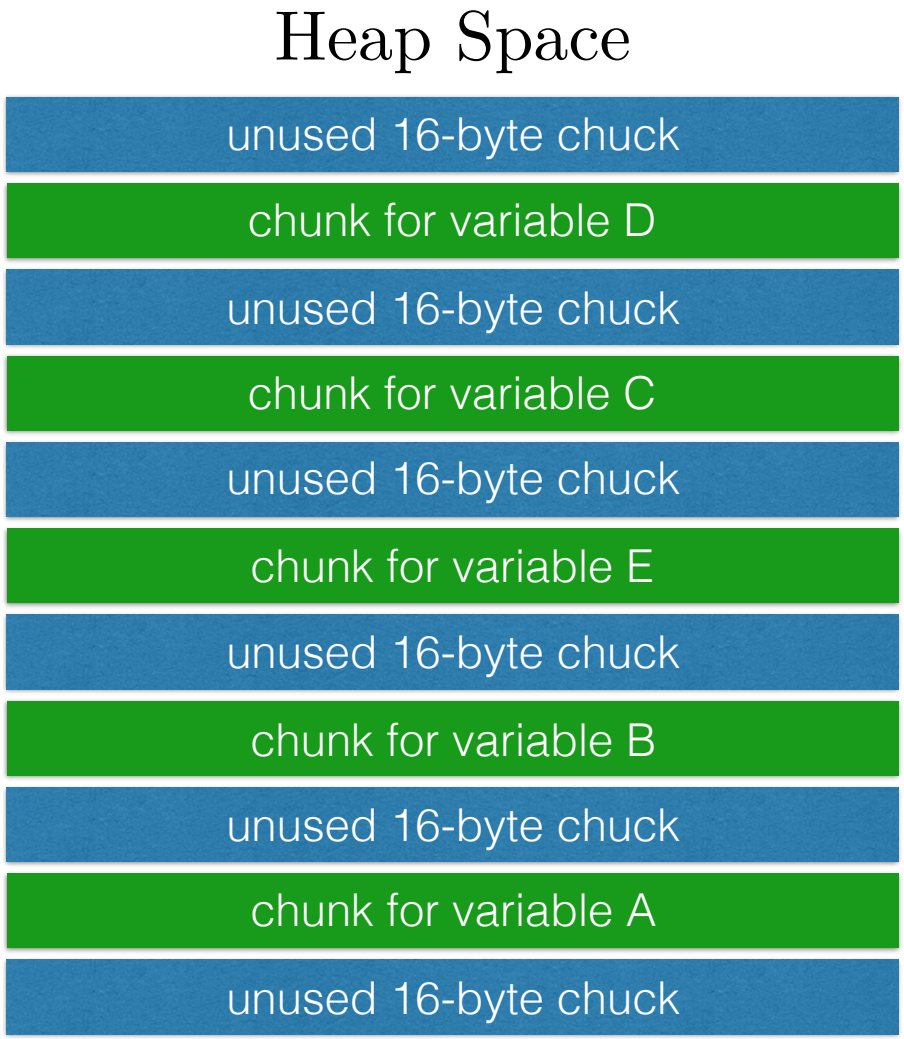
Dynamic Relocation



Dynamic Relocation



Heap Fragmentation



96 bytes of free memory but we can't allocate any chuck larger than 16 bytes

External Fragmentation

Compaction

» The operating system can reorganize the fragmented space so that the free space is contiguous.

» Expensive

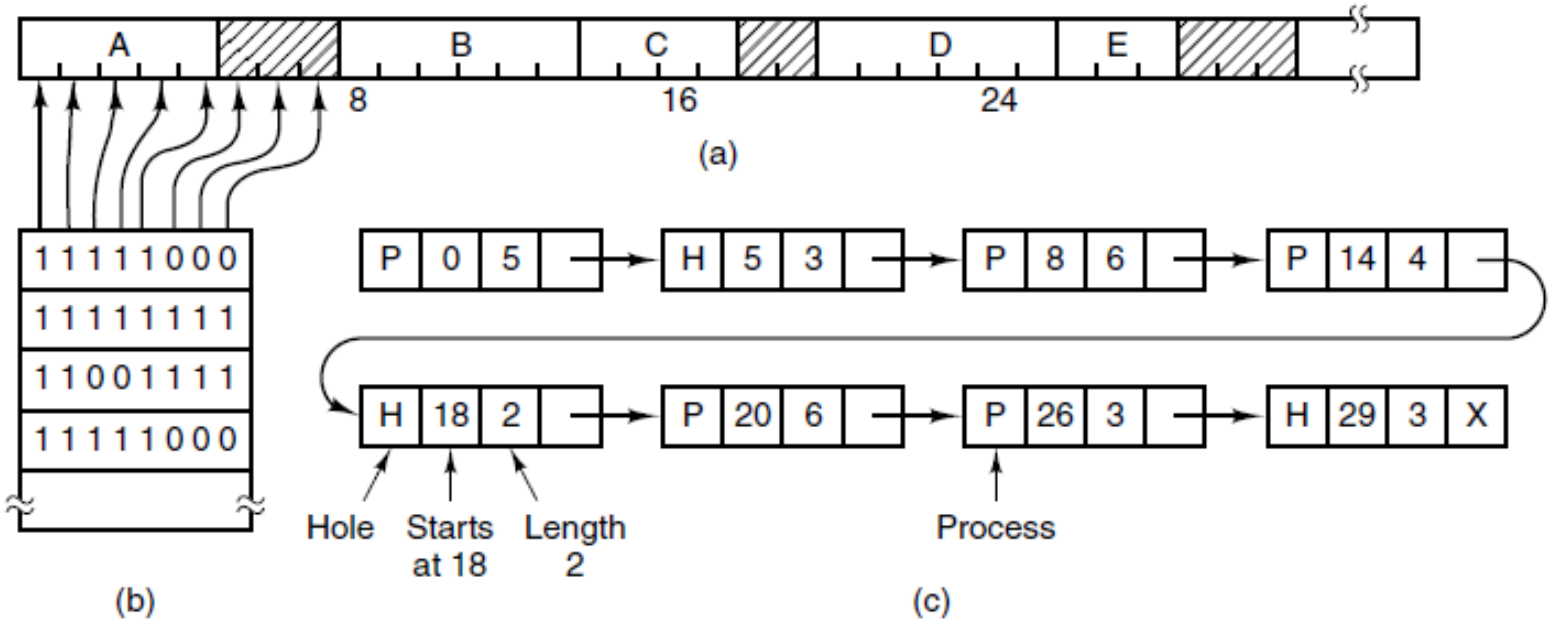
On a 16-GB machine that can copy 8 bytes in 9 sec, it would take 16 sec to compact all of the memory

Managing Free Memory

- Two ways to track memory usage
  - Bit maps
  - Free lists

- Chapter 10: Linux memory allocators such as buddy and slab.

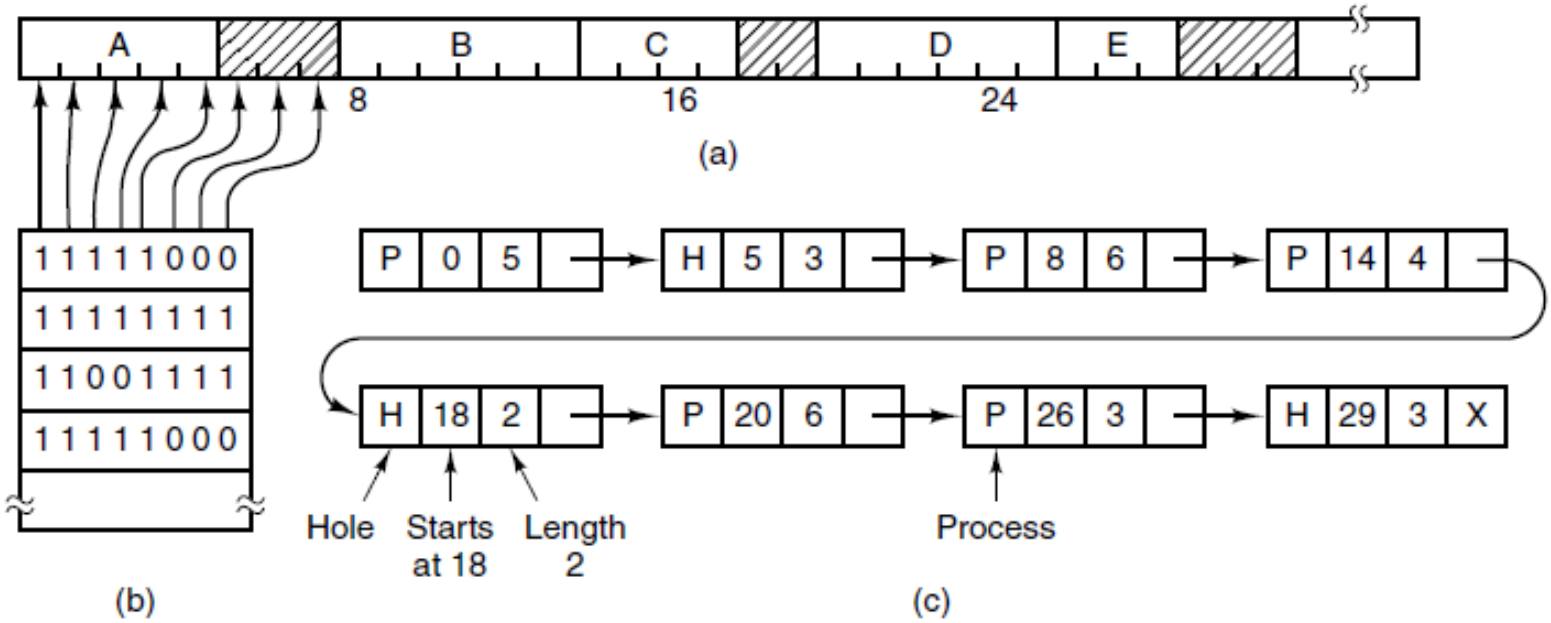
Bitmap



» With bitmap, memory divided into allocation units as small as a few words to a couple kilobytes.

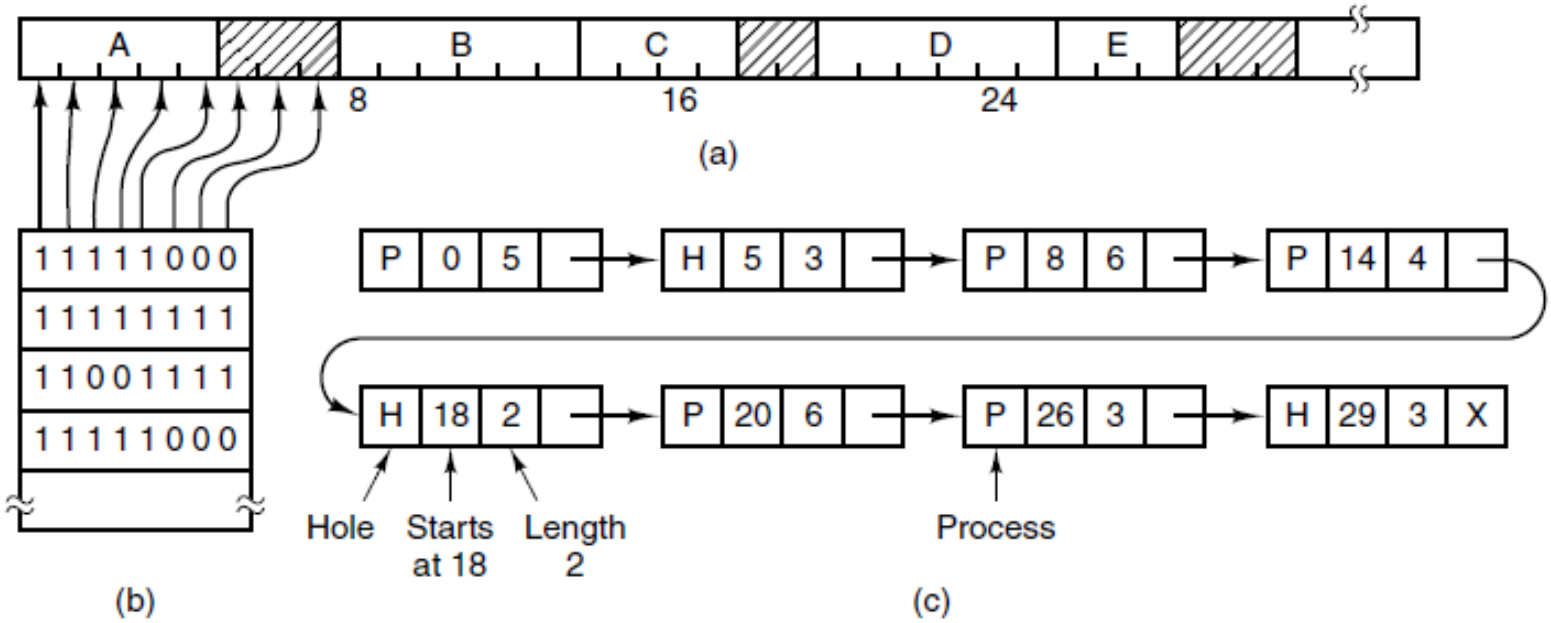
» Each allocation unit corresponds to a bit

Bitmap



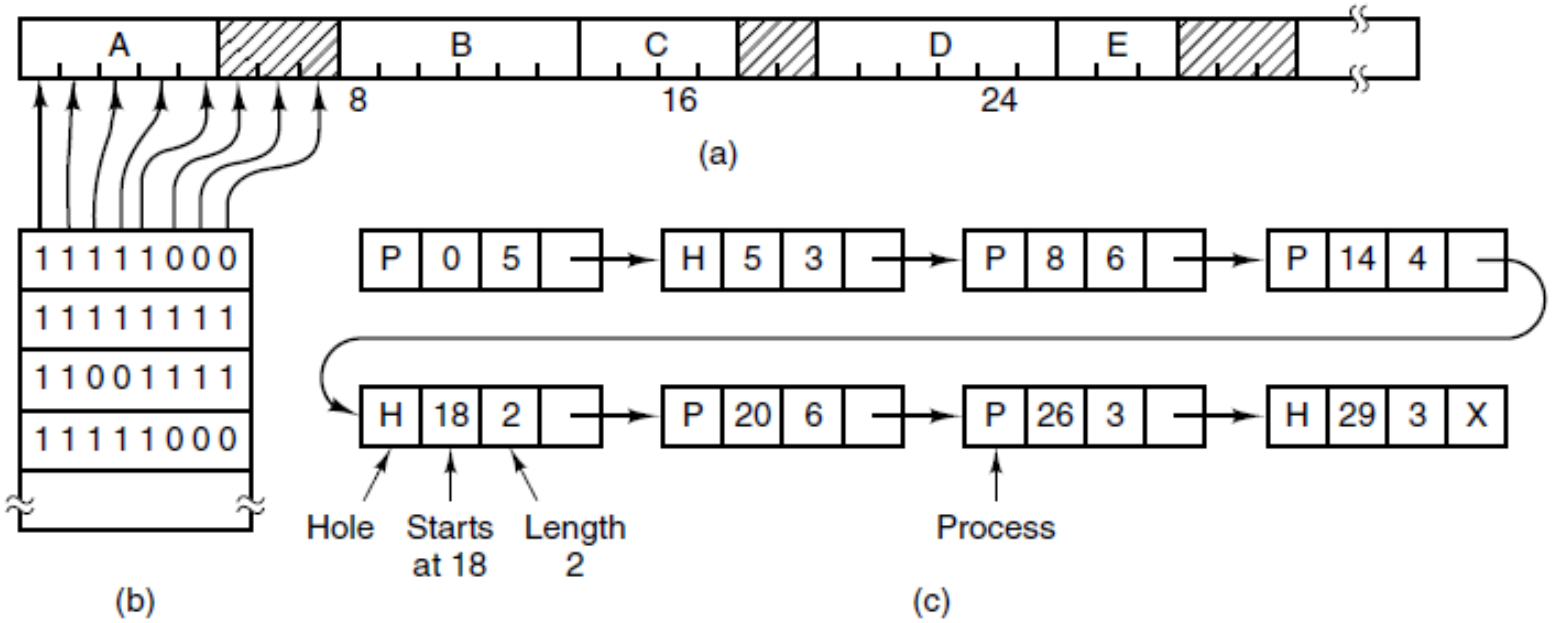
» Problem with bitmap: When bringing in K-units, the memory manager must search the bitmap for a run of k consecutive 0's

Bitmap



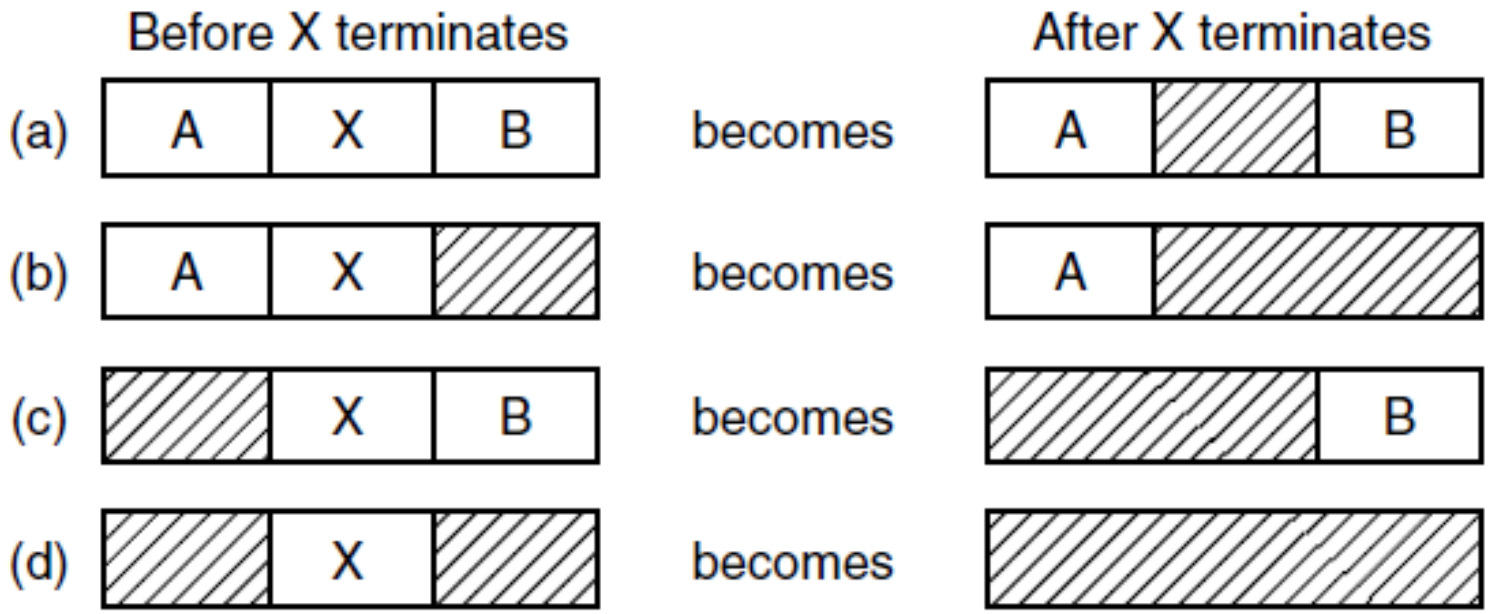
» Problem with bitmap: When bringing in K-units, the memory manager must search the bitmap for a run of k consecutive 0's

Linked List

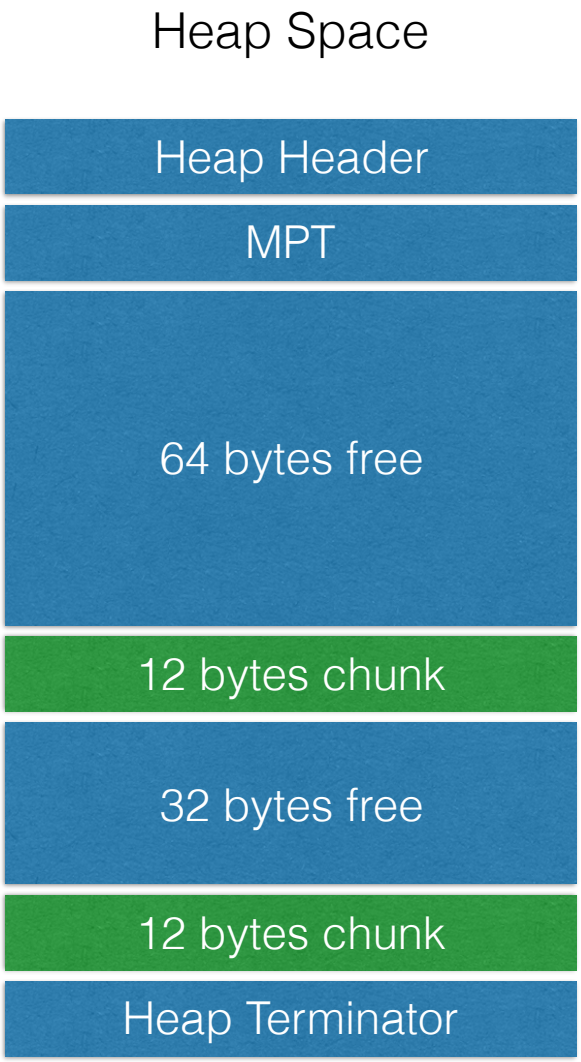


» Linked list: list of allocated and free memory segments.

Linked List


Before X terminates
After X terminates
(a)
A
X
B
becomes
A
B
(b)
A
X
becomes
A
(c)
X
B
becomes
B
(d)
X
becomes

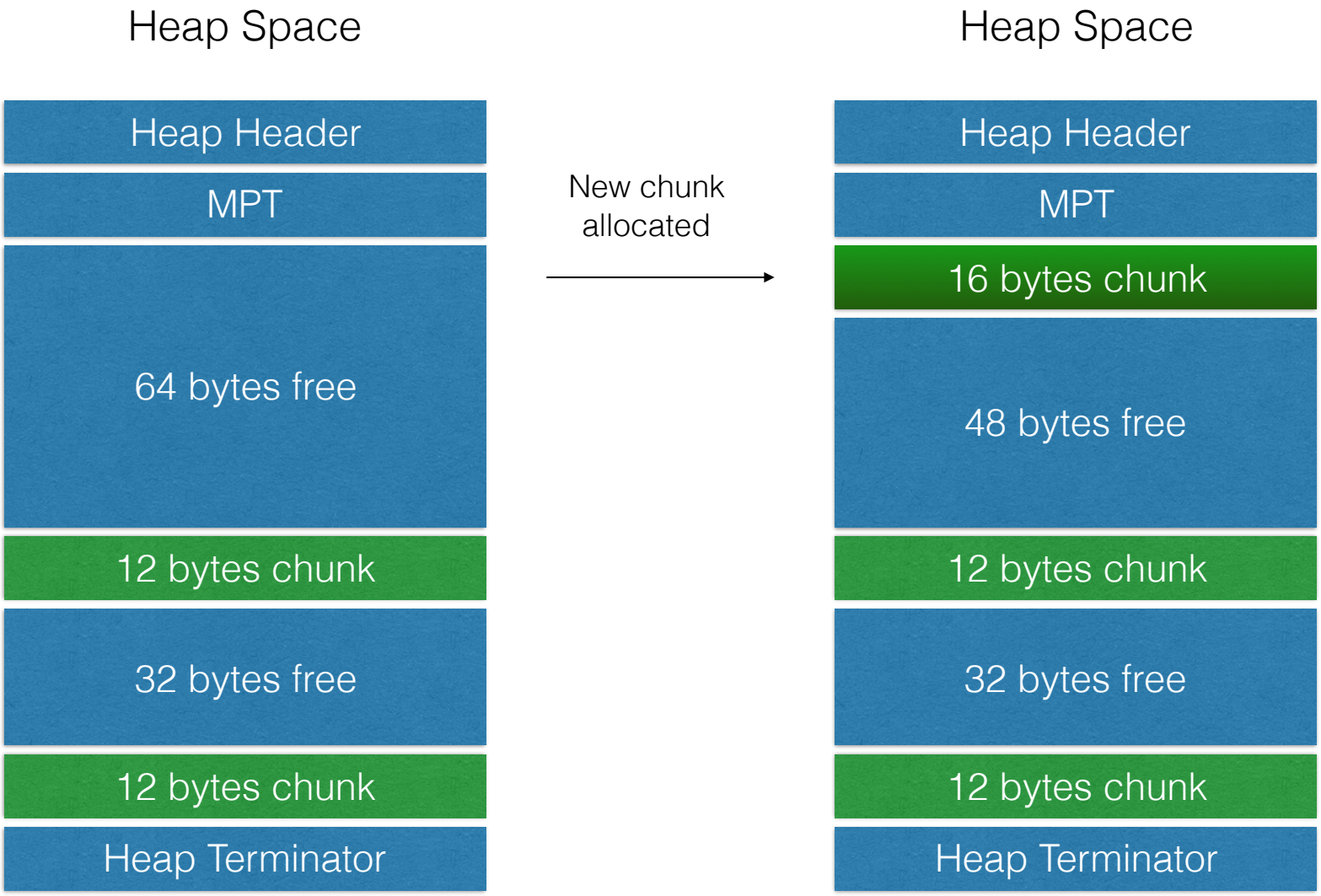
Allocating Memory



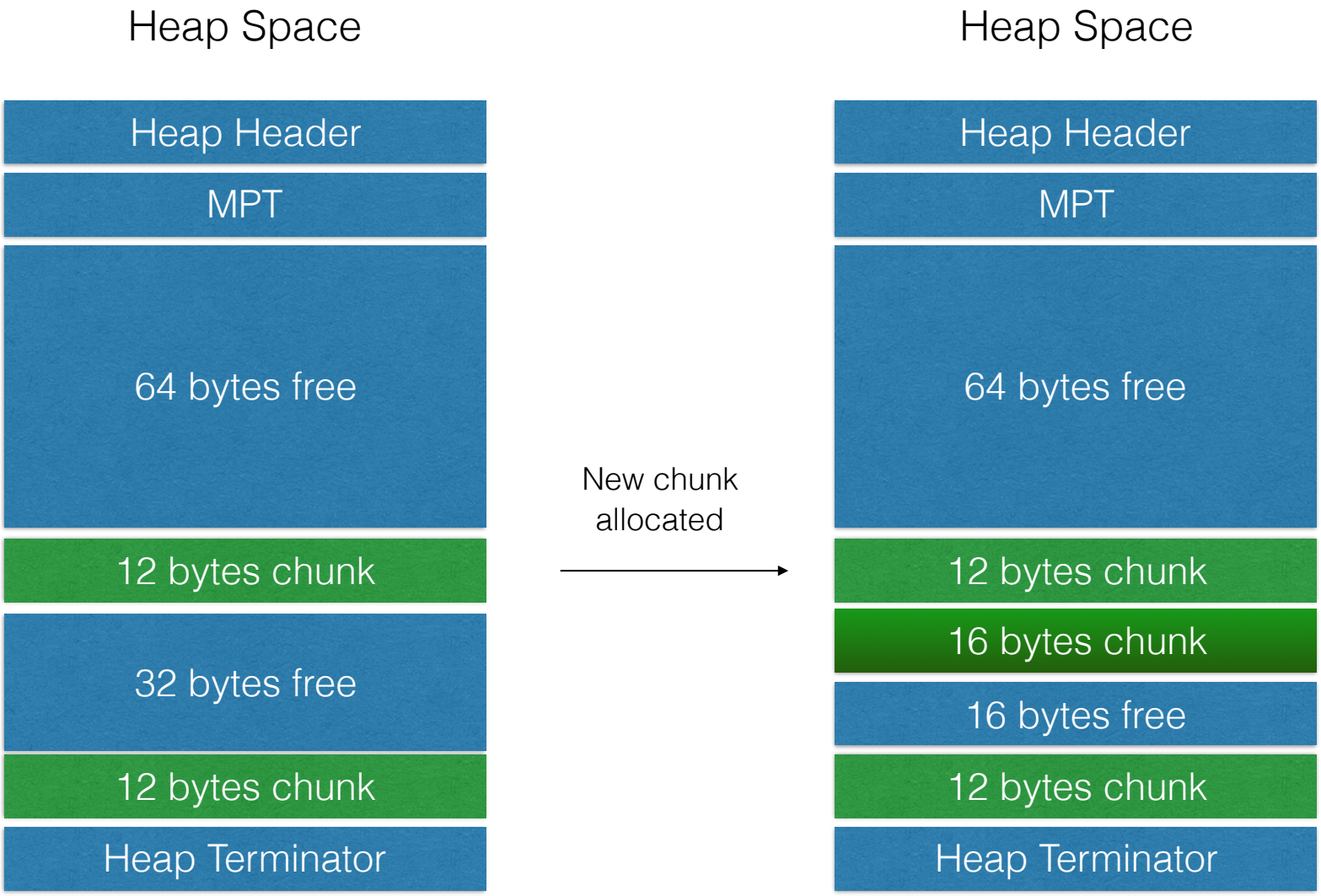
Example: We want to allocate a new 16 byte block.

Which free block does the OS choose?

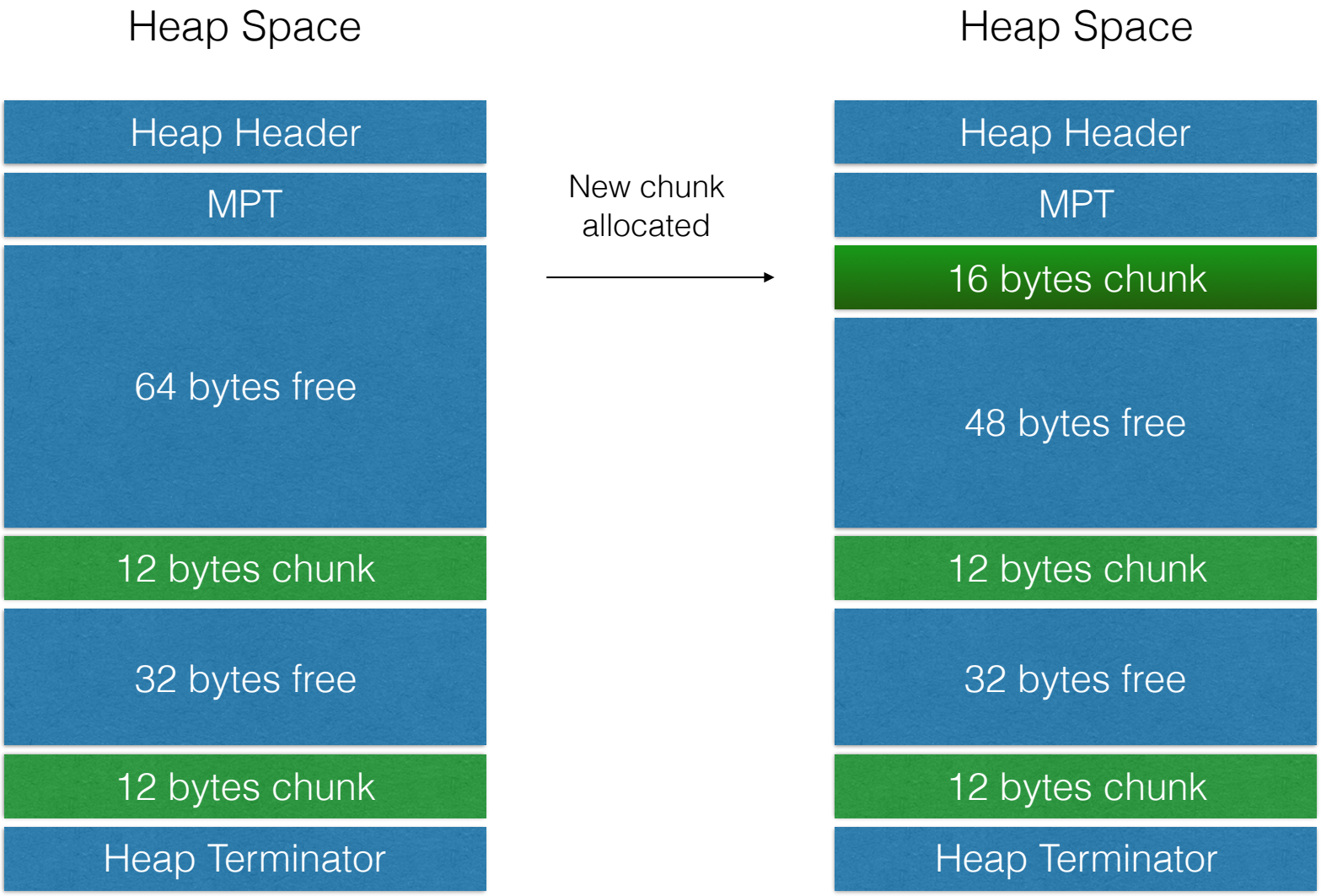
First Fit


Heap Space
Heap Header
MPT
64 bytes free
12 bytes chunk
32 bytes free
12 bytes chunk
Heap Terminator
New chunk allocated
Heap Space
Heap Header
MPT
16 bytes chunk
48 bytes free
12 bytes chunk
32 bytes free
12 bytes chunk
Heap Terminator

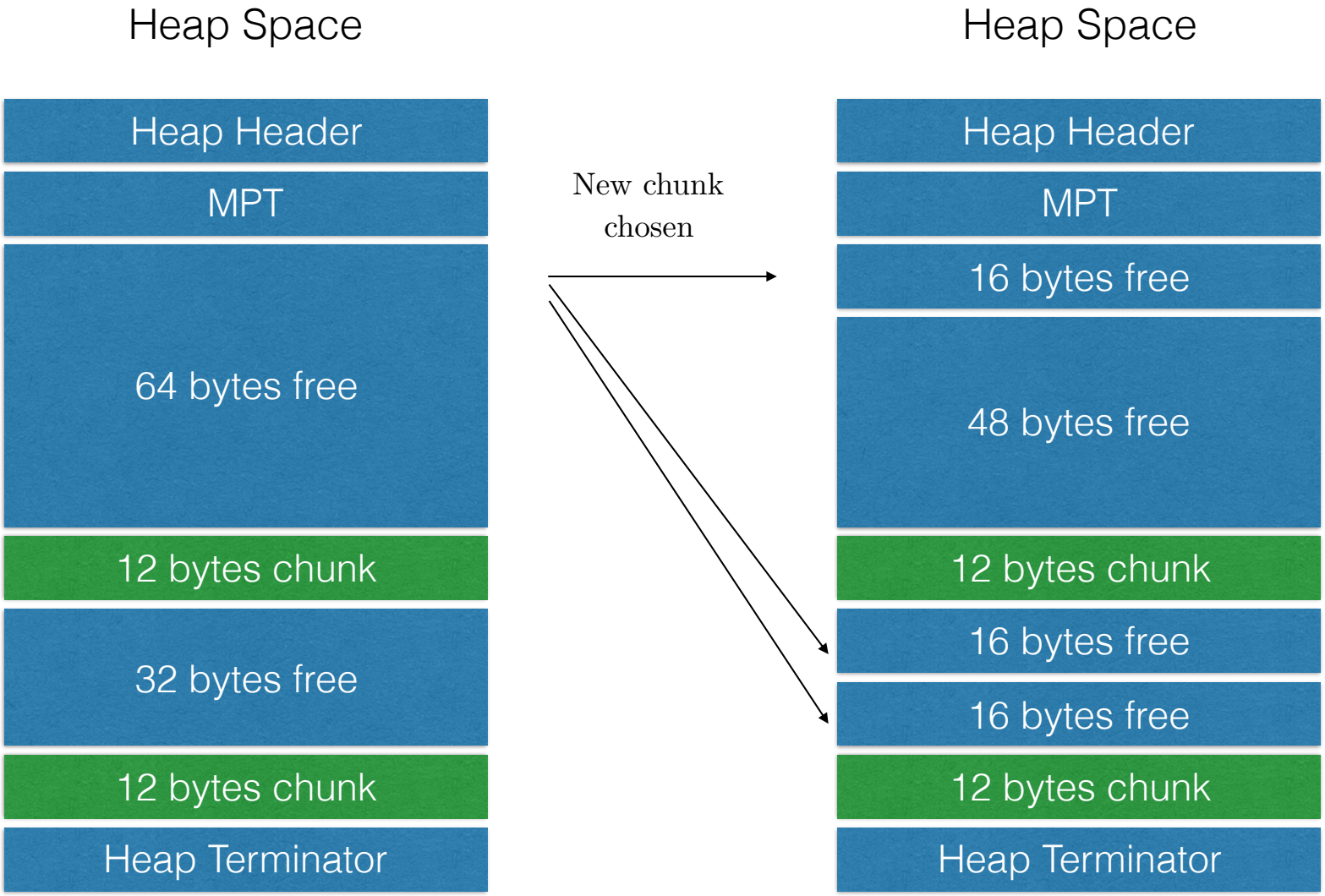
Best Fit


Heap Space
Heap Header
MPT
64 bytes free
12 bytes chunk
32 bytes free
12 bytes chunk
Heap Terminator
New chunk allocated
Heap Space
Heap Header
MPT
64 bytes free
12 bytes chunk
16 bytes chunk
16 bytes free
12 bytes chunk
Heap Terminator

Worst Fit


Heap Space
Heap Header
MPT
64 bytes free
12 bytes chunk
32 bytes free
12 bytes chunk
Heap Terminator
New chunk allocated
Heap Space
Heap Header
MPT
16 bytes chunk
48 bytes free
12 bytes chunk
32 bytes free
12 bytes chunk
Heap Terminator

Quick Fit



Maintain list for common sizes

We need help, but why?

» Multiprocessing causes external fragmentation

» Compaction wastes resources

» Solution – break RAM into fixed parts

» Do not need to be contiguous

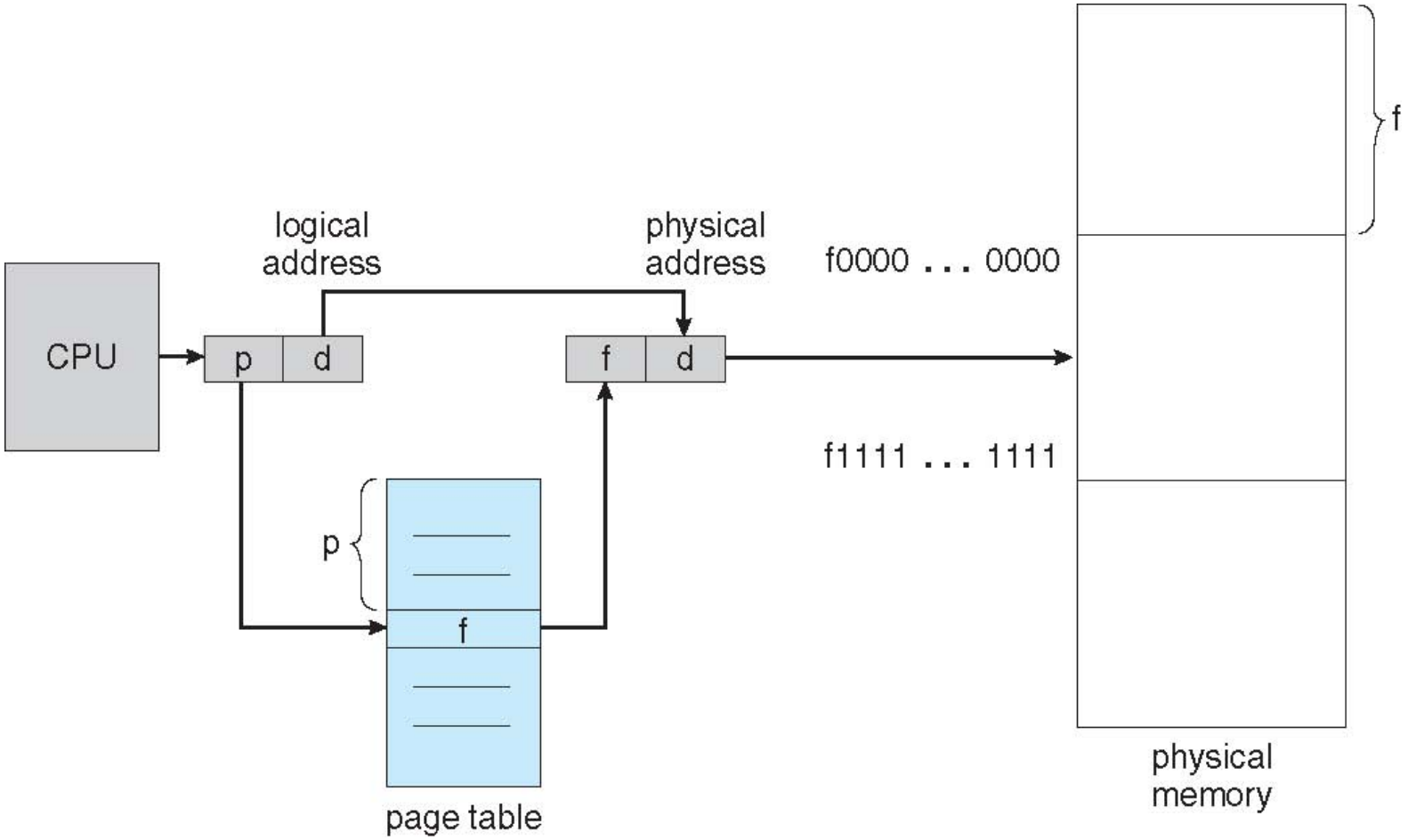
» Map from logical to physical spaces

» Need hardware help

Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8,192 bytes)
  - Commonly 4 KB
- Divide logical memory into blocks of same size called pages
- Keep track of all free frames
- To run a program of size n pages, need to find n free frames and load program
- Set up a page table to translate logical to physical addresses
  - Holds the map to the physical address
- Internal fragmentation

Paging Hardware

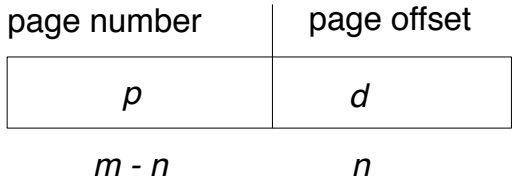


Address Translation Scheme

» Address generated by CPU is divided into:

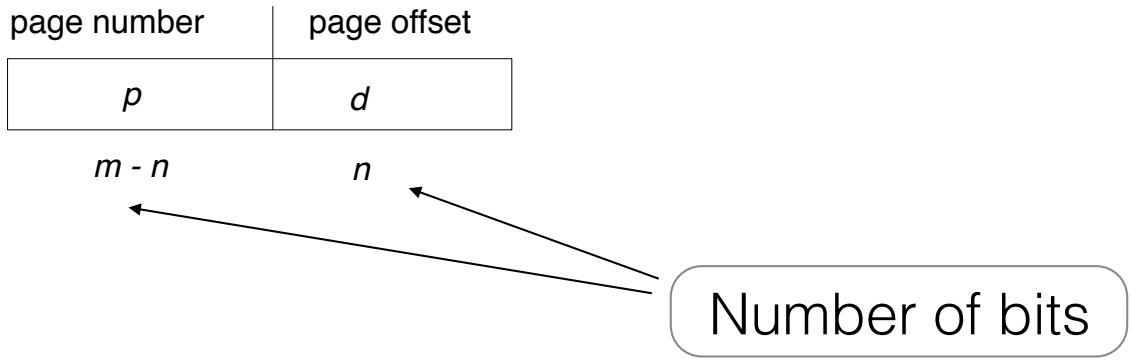
- **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
- **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

- For given logical address space 2^m and page size 2^n



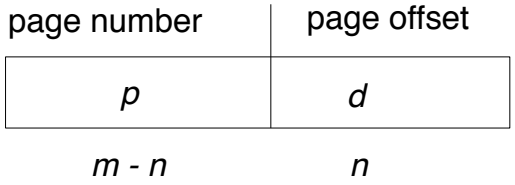
Address Translation Scheme

– For given logical address space 2^m and page size 2^n



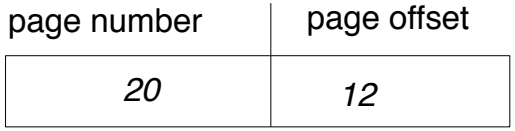
Address Translation Scheme

– For a system with 4 GB of addressable logical address space and a page size of 4 KB, the logical address is:



Page size: 4096 = 2^{12}

Address space: 4294967296 = 2^{32}



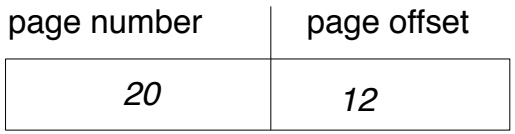
Address Translation Scheme

– Number of pages:

$$4294967296 / 4096 = 1048576$$

» And to verify our numbers:

$$1048576 = 2^{20}$$



Address Translation Scheme

For a system with 12 bit addressing:

1. What is the maximum size of addressable real memory?

If the addresses are split into a page (6 bits) and an offset (displacement, 6 bits)

1. How large is a memory frame?
2. How many entries (maximum) is the page table?

Address Translation Scheme

For a system with 12 bit addressing:

1. What is the maximum size of addressable real memory?

$$2^{12} = 4096 \text{ bytes}$$

Address Translation Scheme

If the addresses are split into a page (6 bits) and an offset (6 bits)

1. How large is a memory frame?

2^6=64 bytes

2. How many entries (maximum) is the page table?

2^6 = 64 pages