2.6 Operating System Design

- There is no unique process for designing an OS, but some approaches have proven successful. Design starts by defining goals and specifications

- At highest level, design is affected by choice of **hardware** and **type of system**:
  - batch, single user, or multi user
  - Distributed?
  - real time?

- Next level are requirements for **user** goals and **system** goals
  - User goals – operating system should be
    - Easy to use
    - Responsive
  - System goals – operating system should be
    - Easy to design, implement, and maintain
    - Flexible and efficient
    - Reliable and error-free

Operating System Design

- An important design choice may be to separate

  Mechanism: ***How*** to do it?

  Policy: ***What*** specifically will be done?

  - e.g.: The OS timer construct for preempting processes is a mechanism, but the exact tick time is a policy.
- The separation of policy from mechanism is a very important principle, it allows maximum **flexibility** if policy decisions are to be changed later
  - e.g.: A scheduling mechanism may be made general-purpose to allow various policy setups such as time-sharing, batch, real-time etc.
- In some systems, both mechanisms and policy are encoded to enforce a **global look** such as in Windows and Mac OS X, but not in Unix (e.g. different window managers such as KDE, GNOME, etc.)

Operating System Implementation

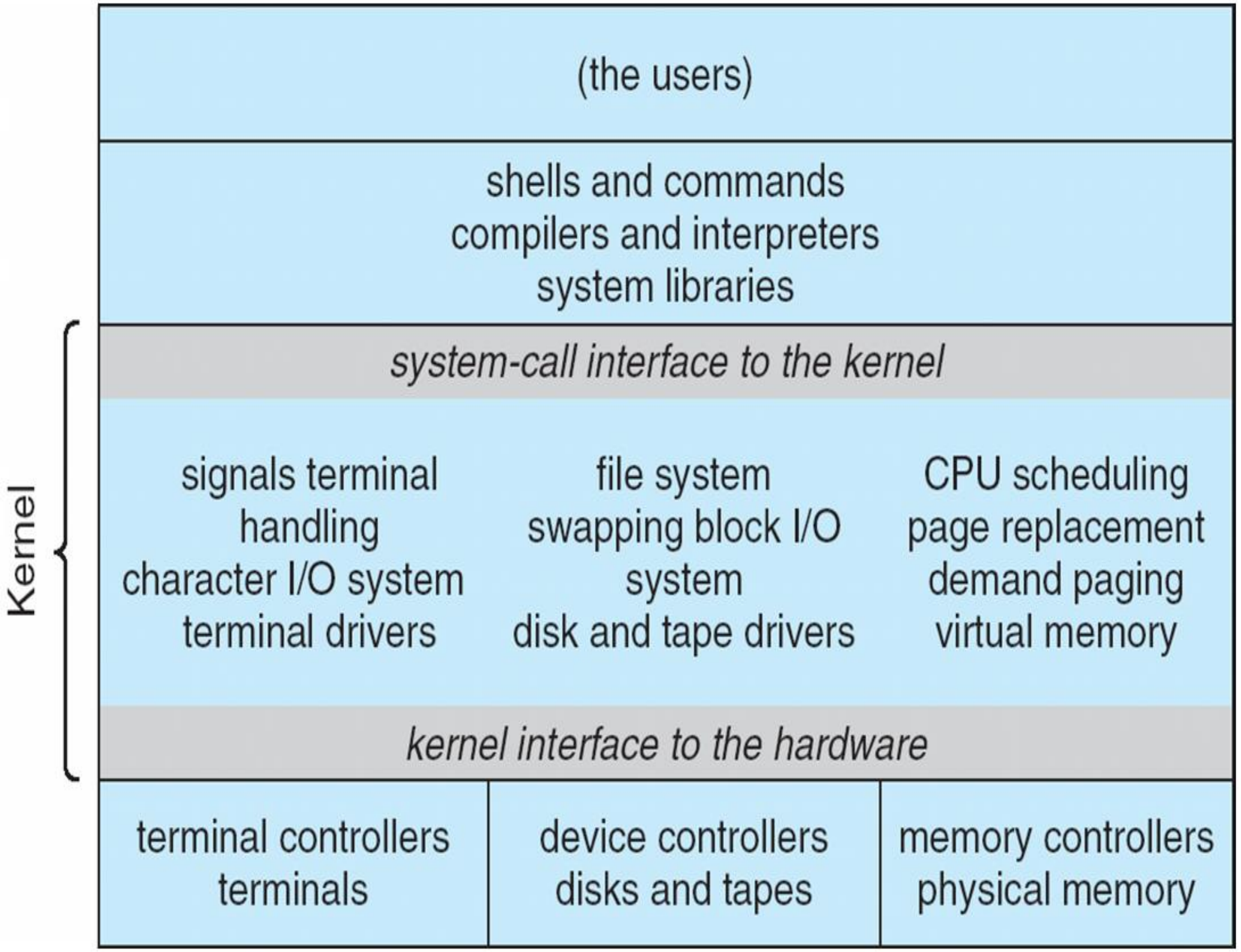
- Early OSes in assembly language (efficient but difficult to code/maintain)
- C, C++ (less memory-efficient, less performance but easier to code/maintain)
- Now a mix of languages
  - **Lowest levels** in assembly, for hardware/architecture-dependent code and for time critical areas, e.g. **within** interrupt handlers, device and memory managers, and schedulers
  - Main body of **kernel** is written in C. Improving the data structures and algorithms has more impact than coding in a lower-level language.
  - **Systems programs** in C, C++, scripting languages like PERL, Python, shell scripts

2.7 Operating System Structure

- A general-purpose OS is very large and complex program
- Various ways to structure an OS:
  - Simple (monolithic) structure – e.g. MS-DOS
  - More complex – e.g. UNIX
  - Layered – provides abstraction levels
  - Microkernel – e.g. Mach and Minix

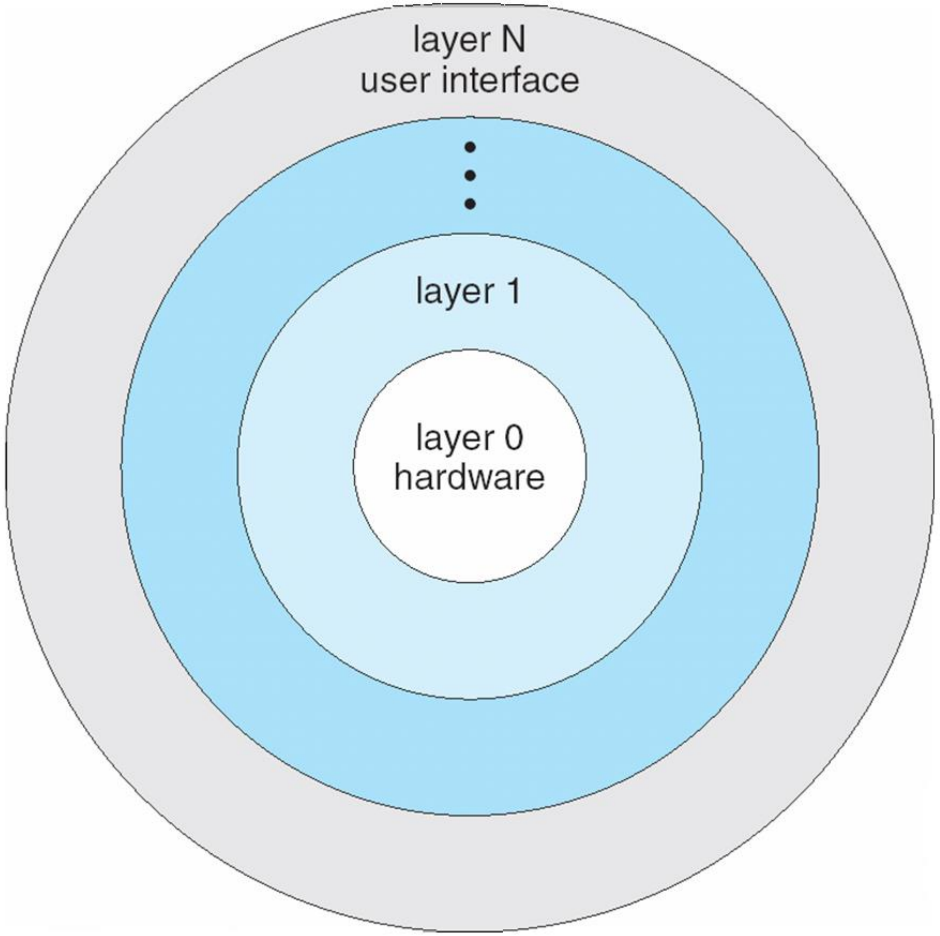
Simple Structure -- The original Unix

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions
    - That's large number of functions for a monolithic one-level system.
  - Beyond simple, but not fully layered.



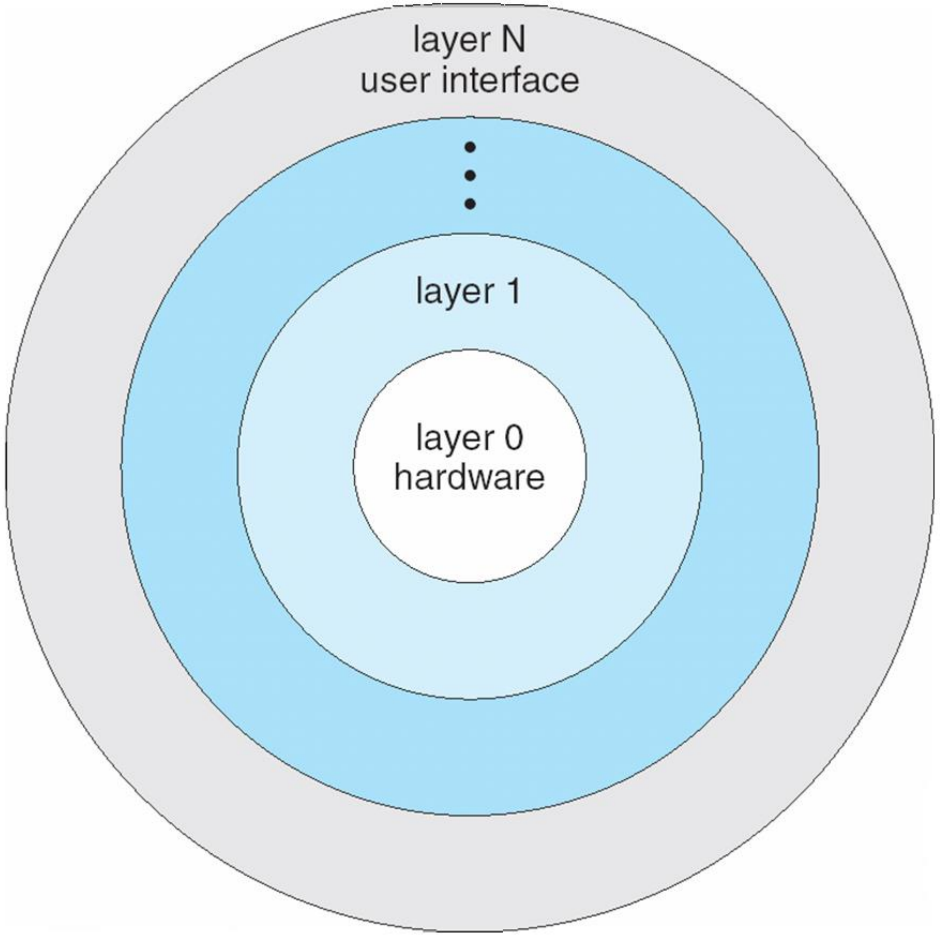
Layered (hierarchial) Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of the layer underneath it.



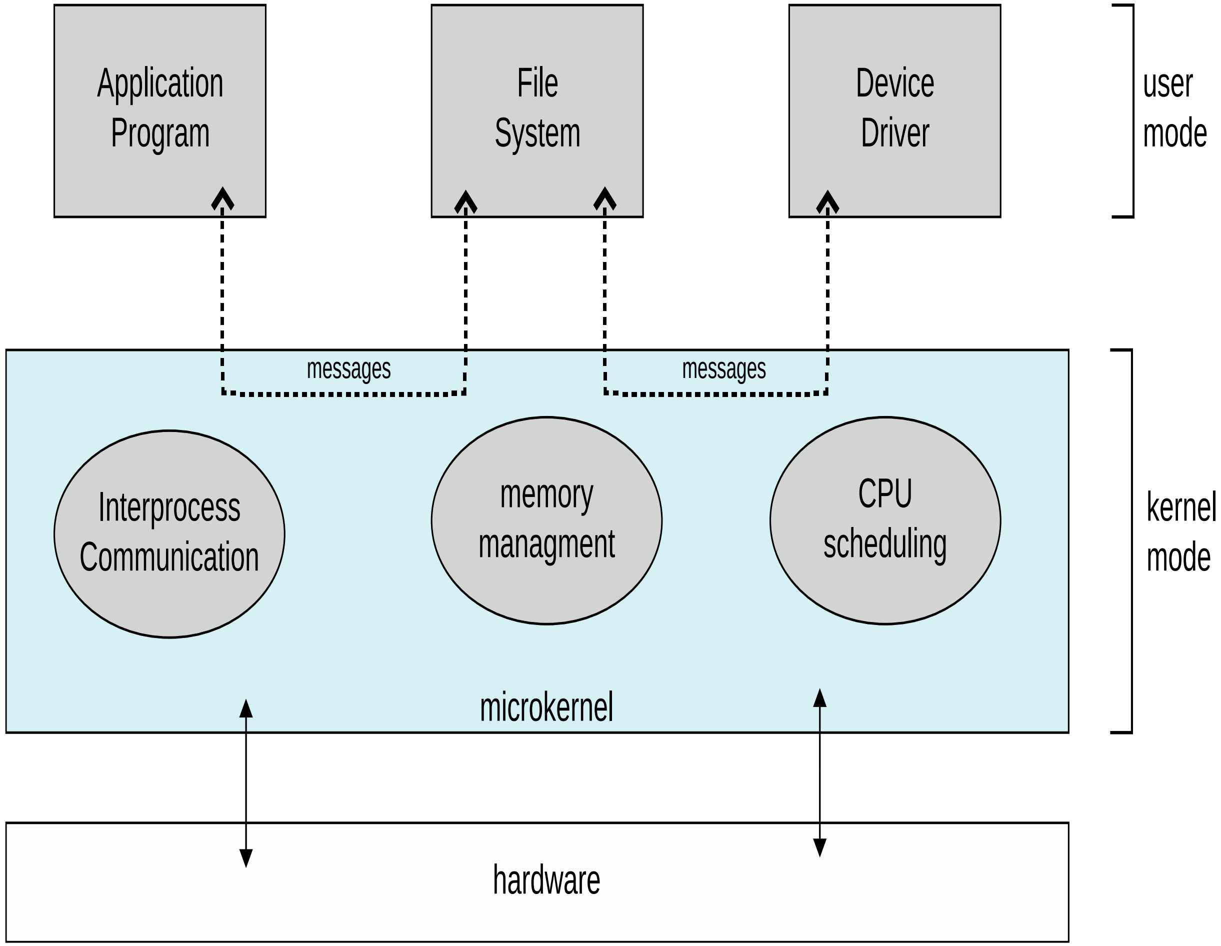
Layered Approach (cont.)

- **Advantages:** Ease of implementation and debugging.
  - The lowest layer may be debugged first (without debugging the higher layers), and once that's done, the next layer can then be debugged, and so on.
- **Disadvantages:** (caused the layered approach to fall out of favor)
  - Difficulty in realizing layered levels (e.g. both a memory manager and disk manager may want use each other's services, same for scheduler/disk manger)
  - Less efficient – a call from upper layer propagates through many lower layers.



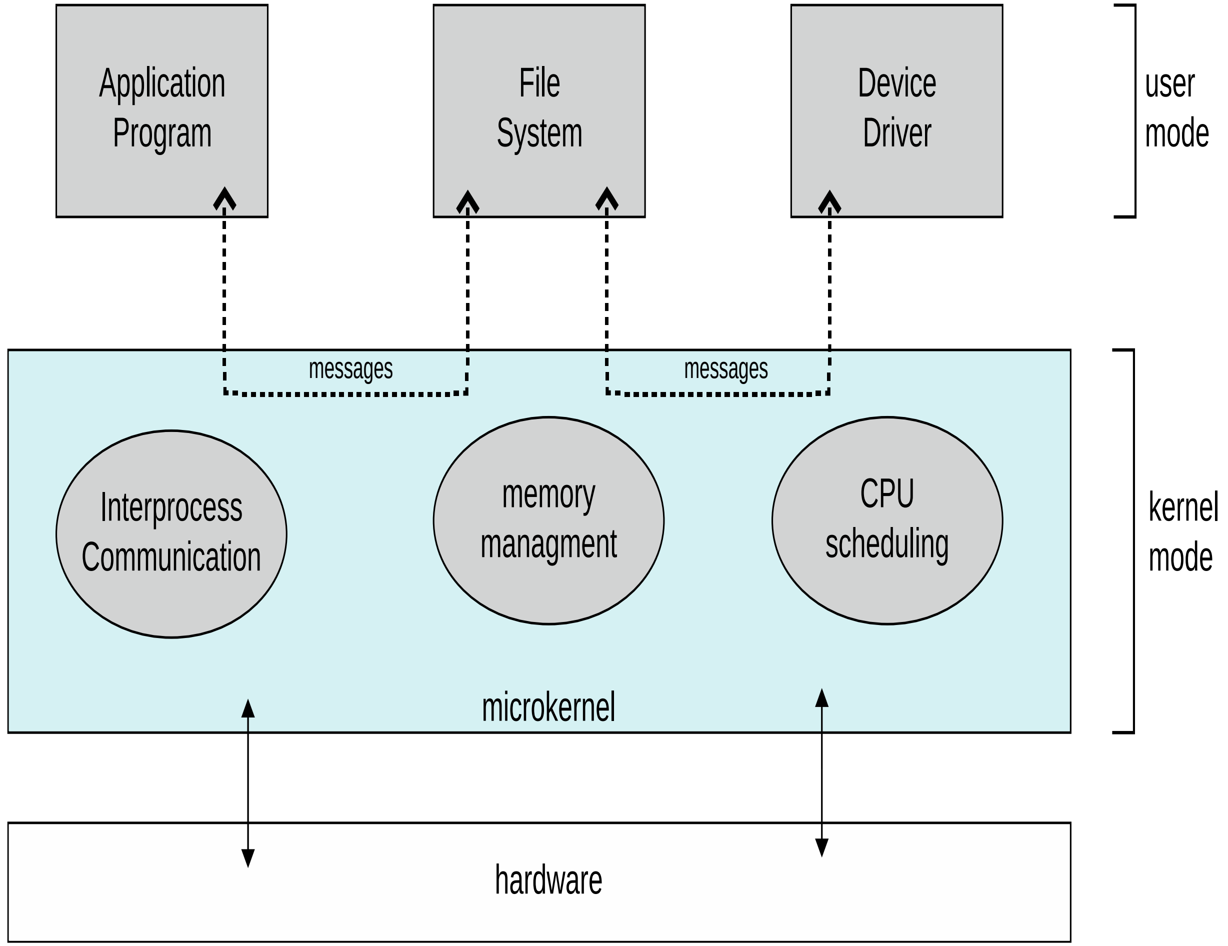
Microkernels

- Moves as much from the kernel into user space
- **Mach** example of **microkernel**
  - Developed in mid 1980's @ Carnegie Mellon university.
  - Maps Unix system calls to messages sent to appropriate user level services
  - Mac OS X kernel (**Darwin**) partly based on Mach
- Micro-kernel provides minimal process and memory management, in addition to a communication facility.
- Communication takes place between user modules using **message passing** via the microkernel.



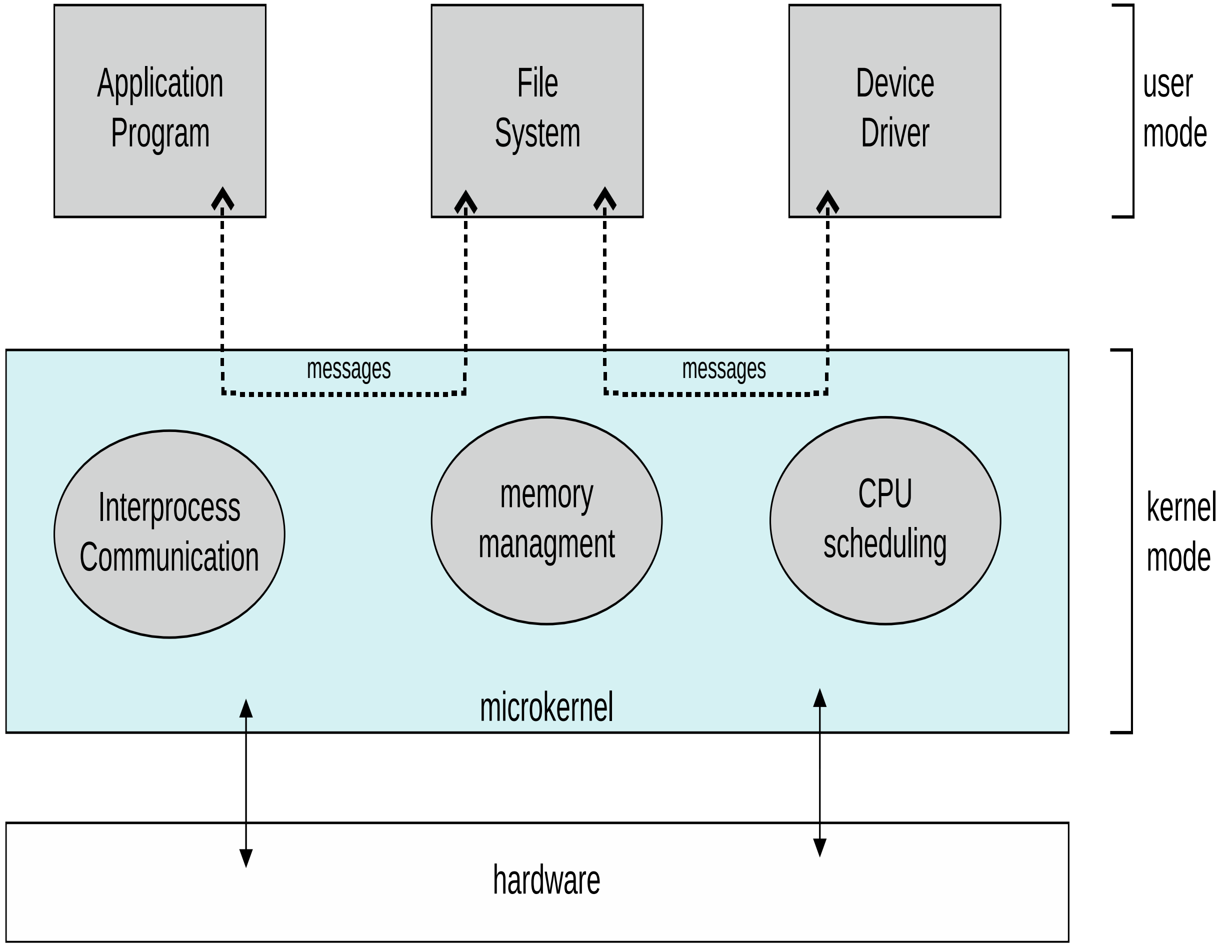
Microkernels – cont.

- Advantages:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Disadvantages:
  - Performance overhead of user space to kernel space, as well as user space to user space communications



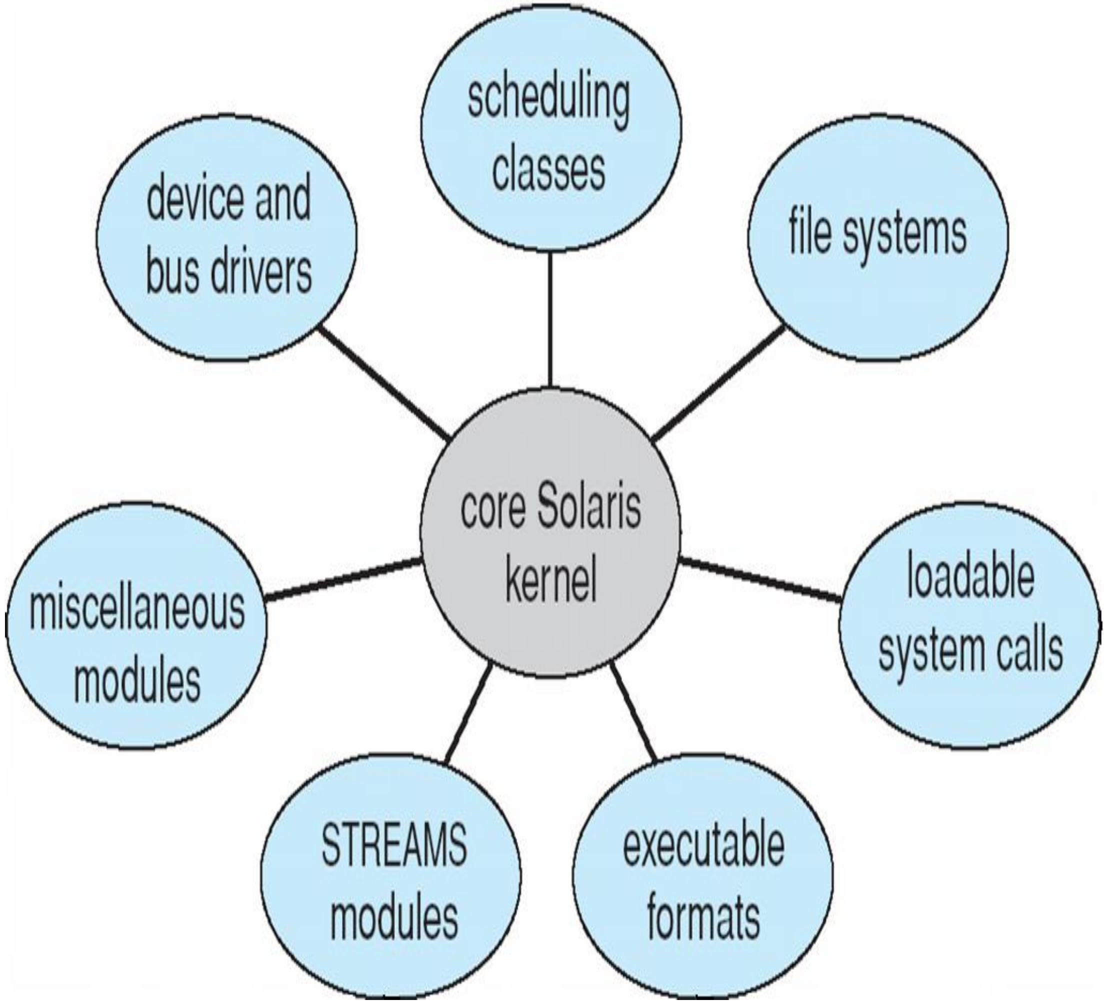
Microkernels – cont.

- Other examples:
  - Tru64 (aka Digital Unix)
  - Minix
  - **QNX Neutrino**,
    - An embedded real-time OS.
    - Kernel only handles process scheduling, memory management, IPC, low level network comm. and H/W interrupts.
    - The Inter-process communication mechanism allows a process waiting for a message to be immediately invoked when a message is sent to it, without invoking the scheduler.
    - IPC messages are sent according the priority of the receiving process
    - Hence tight coupling between the scheduler and IPC.



Subsystems and Loadable Modules

- Many modern operating systems implement **subsystems** and **loadable kernel modules**
  - The kernel has a set of core components/subsystems that are linked (at compile time or load time) to additional services via modules.
  - Subsystems and modules talk to each others over known interfaces
  - Modules may be loaded as needed within the kernel (preferred, as opposed to compile-time linking)
- Overall, similar to layers but with more flexibility
  - Linux, Solaris, Mac OS X, Windows, etc
- The solaris system shown has 7 loadable modules.
- Linux has loadable modules primarily for device drivers and file systems.

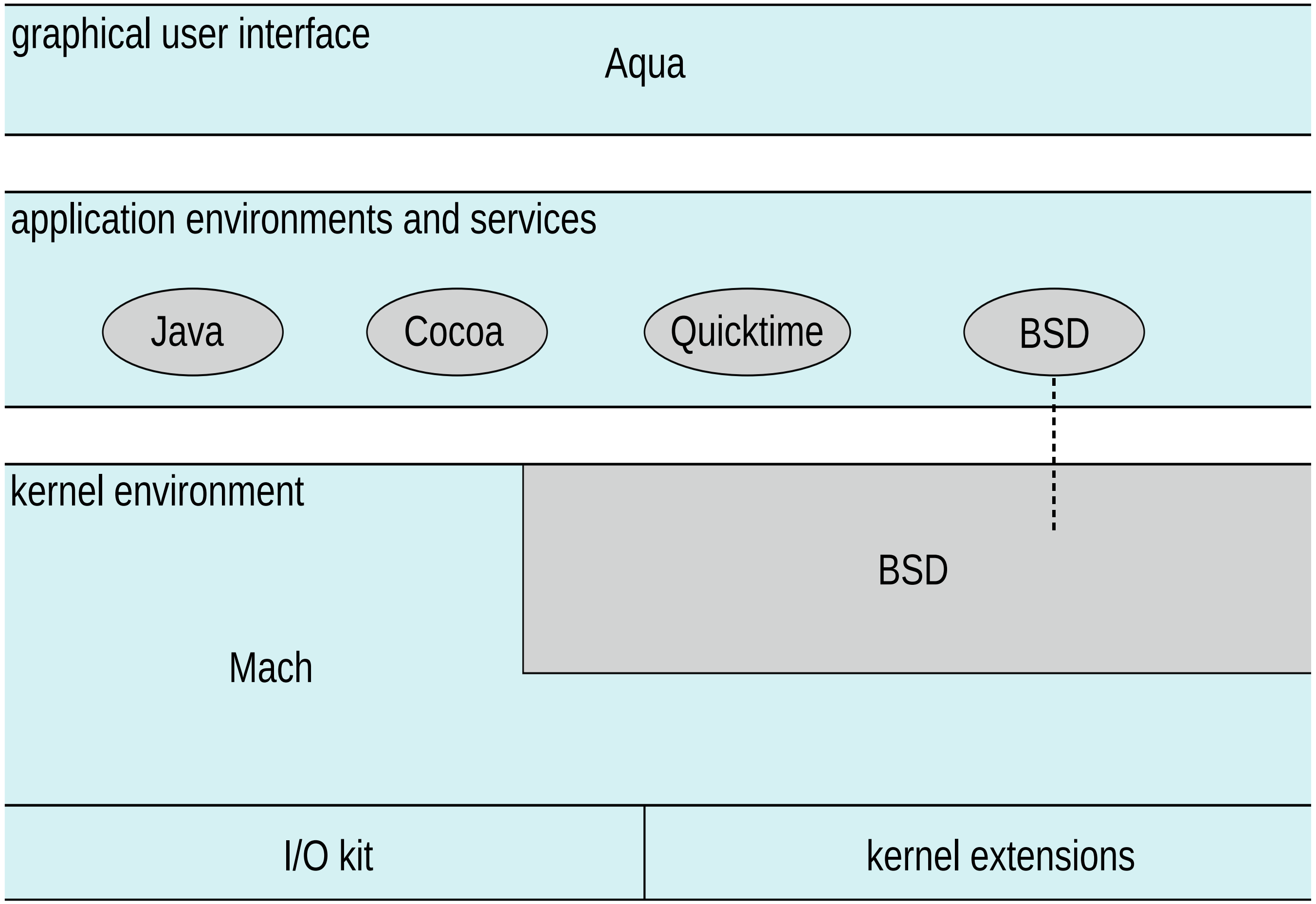


Hybrid Systems

- Most modern operating systems are actually not one pure model
  - Hybrid combines multiple approaches to address performance, security, usability needs
  - **Linux and Solaris kernels** have kernel components that run in the same address space, **plus modular** for dynamic loading of functionality
    - Note: Solaris is a Unix OS developed by Sun Microsystems (currently Oracle)
  - **Windows** has subsystems and modules, but it retains some **microkernel-like** behavior since it has different subsystems running as user-mode processes (referred to as personalities). At the same time it provides dynamically loadable kernel modules.

Mac OS X Structure

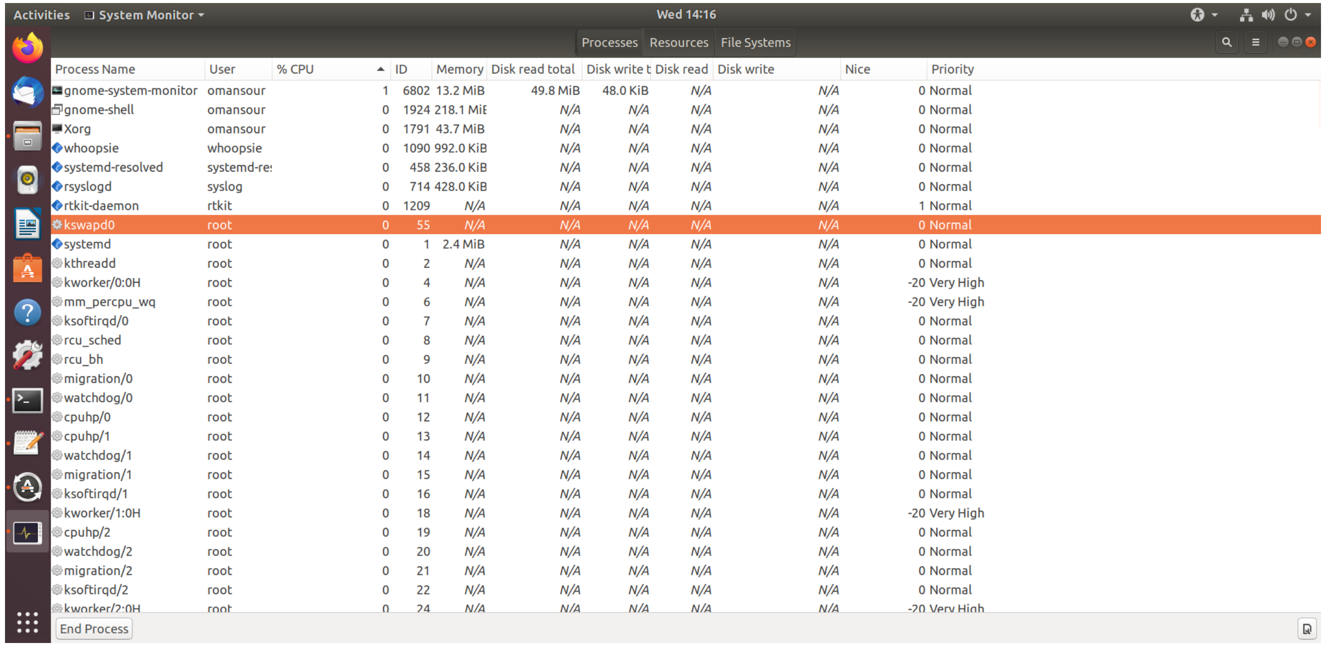
- Apple Mac OS X has a hybrid, layered structure.
- **Aqua** UI
- **Cocoa** programming environment for objective-C
- Shown is kernel consisting of:
  - Mach microkernel for memory management, IPC and scheduling.
  - BSD Unix for CLI, networking, file systems and POSIX APIs (including pthreads).
  - I/O kit for device driver development.
  - Dynamically loadable modules (called **kernel extensions**)



Operating-System Debugging

- **Debugging** is finding and fixing errors, or **bugs**
- OS generate **log files** containing error information
- Failure of an application can generate **core dump** file capturing memory of the process.
  - Location may be found or modified by editing the file proc/sys/kernel/core_pattern
- Operating system failure can generate **crash dump** file containing kernel memory
- Beyond crashes, performance tuning can optimize system performance
  - Sometimes using ***trace listings*** of activities, recorded for analysis
  - **Profiling** is periodic sampling of instruction pointer (i.e. program counter, PC) to look for statistical trends. **perf** is a userspace utility that is accessed from the command line and provides a number of subcommands; it is capable of statistical profiling of the entire system (both kernel and user code).

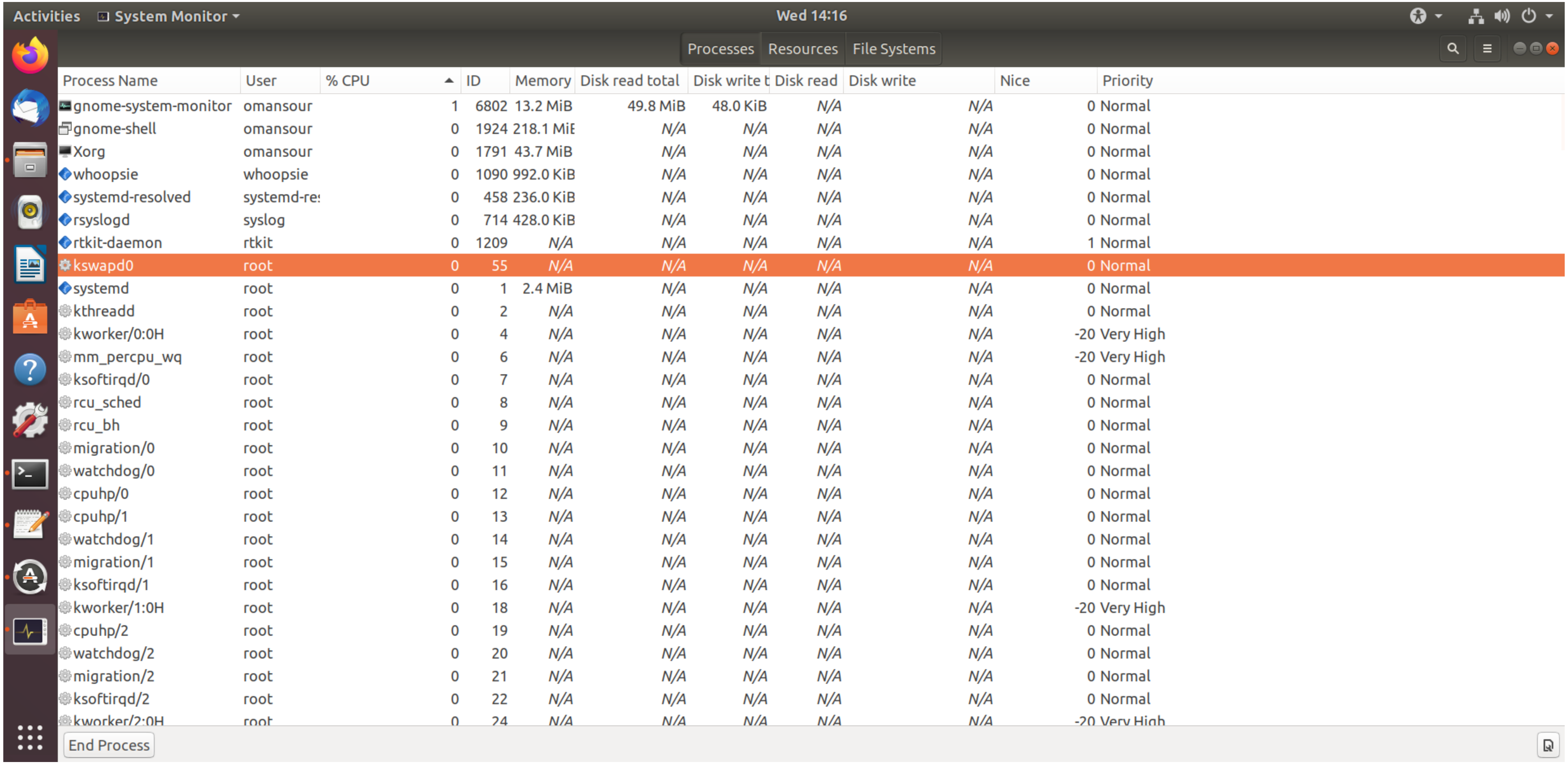
Performance Tuning



Process Name	User	% CPU	ID	Memory	Disk read total	Disk write t	Disk read	Disk write	Nice	Priority
gnome-system-monitor	omansour	1	6802	13.2 MiB	49.8 MiB	48.0 KiB	N/A	N/A	0	Normal
gnome-shell	omansour	0	1924	218.1 MiE	N/A	N/A	N/A	N/A	0	Normal
Xorg	omansour	0	1791	43.7 MiB	N/A	N/A	N/A	N/A	0	Normal
whoopsie	whoopsie	0	1090	992.0 KiB	N/A	N/A	N/A	N/A	0	Normal
systemd-resolved	systemd-re:	0	458	236.0 KiB	N/A	N/A	N/A	N/A	0	Normal
rsyslogd	syslog	0	714	428.0 KiB	N/A	N/A	N/A	N/A	0	Normal
rtkit-daemon	rtkit	0	1209	N/A	N/A	N/A	N/A	N/A	1	Normal
kswapd0	root	0	55	N/A	N/A	N/A	N/A	N/A	0	Normal
systemd	root	0	1	2.4 MiB	N/A	N/A	N/A	N/A	0	Normal
kthreadd	root	0	2	N/A	N/A	N/A	N/A	N/A	0	Normal
kworker/0:0H	root	0	4	N/A	N/A	N/A	N/A	N/A	-20	Very High
mm_percpu_wq	root	0	6	N/A	N/A	N/A	N/A	N/A	-20	Very High
ksoftirqd/0	root	0	7	N/A	N/A	N/A	N/A	N/A	0	Normal
rcu_sched	root	0	8	N/A	N/A	N/A	N/A	N/A	0	Normal
rcu_bh	root	0	9	N/A	N/A	N/A	N/A	N/A	0	Normal
migration/0	root	0	10	N/A	N/A	N/A	N/A	N/A	0	Normal
watchdog/0	root	0	11	N/A	N/A	N/A	N/A	N/A	0	Normal
cpuhp/0	root	0	12	N/A	N/A	N/A	N/A	N/A	0	Normal
cpuhp/1	root	0	13	N/A	N/A	N/A	N/A	N/A	0	Normal
watchdog/1	root	0	14	N/A	N/A	N/A	N/A	N/A	0	Normal
migration/1	root	0	15	N/A	N/A	N/A	N/A	N/A	0	Normal
ksoftirqd/1	root	0	16	N/A	N/A	N/A	N/A	N/A	0	Normal
kworker/1:0H	root	0	18	N/A	N/A	N/A	N/A	N/A	-20	Very High
cpuhp/2	root	0	19	N/A	N/A	N/A	N/A	N/A	0	Normal
watchdog/2	root	0	20	N/A	N/A	N/A	N/A	N/A	0	Normal
migration/2	root	0	21	N/A	N/A	N/A	N/A	N/A	0	Normal
ksoftirqd/2	root	0	22	N/A	N/A	N/A	N/A	N/A	0	Normal
kworker/2:0H	root	0	24	N/A	N/A	N/A	N/A	N/A	-20	Very High

- Improve performance by removing bottlenecks
- OS must provide means of computing and displaying measures of system behavior
- Windows: Task manager
- Linux (Gnome or KDE): “system monitor”

Performance Tuning

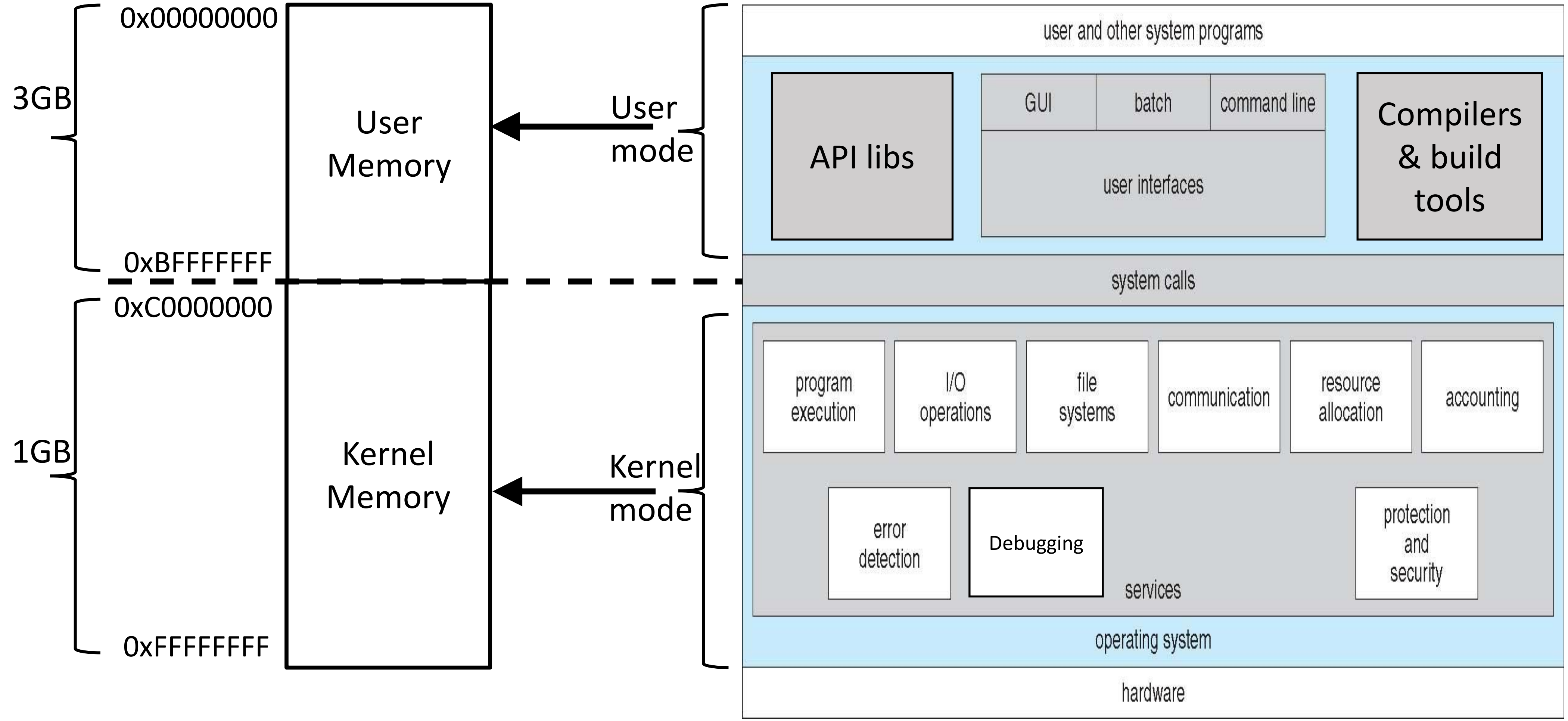


Process Name	User	% CPU	ID	Memory	Disk read total	Disk write t	Disk read	Disk write	Nice	Priority
gnome-system-monitor	omansour	1	6802	13.2 MiB	49.8 MiB	48.0 KiB	N/A	N/A	0	Normal
gnome-shell	omansour	0	1924	218.1 MiE	N/A	N/A	N/A	N/A	0	Normal
Xorg	omansour	0	1791	43.7 MiB	N/A	N/A	N/A	N/A	0	Normal
whoopsie	whoopsie	0	1090	992.0 KiB	N/A	N/A	N/A	N/A	0	Normal
systemd-resolved	systemd-re:	0	458	236.0 KiB	N/A	N/A	N/A	N/A	0	Normal
rsyslogd	syslog	0	714	428.0 KiB	N/A	N/A	N/A	N/A	0	Normal
rtkit-daemon	rtkit	0	1209	N/A	N/A	N/A	N/A	N/A	1	Normal
kswapd0	root	0	55	N/A	N/A	N/A	N/A	N/A	0	Normal
systemd	root	0	1	2.4 MiB	N/A	N/A	N/A	N/A	0	Normal
kthreadd	root	0	2	N/A	N/A	N/A	N/A	N/A	0	Normal
kworker/0:0H	root	0	4	N/A	N/A	N/A	N/A	N/A	-20	Very High
mm_percpu_wq	root	0	6	N/A	N/A	N/A	N/A	N/A	-20	Very High
ksoftirqd/0	root	0	7	N/A	N/A	N/A	N/A	N/A	0	Normal
rcu_sched	root	0	8	N/A	N/A	N/A	N/A	N/A	0	Normal
rcu_bh	root	0	9	N/A	N/A	N/A	N/A	N/A	0	Normal
migration/0	root	0	10	N/A	N/A	N/A	N/A	N/A	0	Normal
watchdog/0	root	0	11	N/A	N/A	N/A	N/A	N/A	0	Normal
cpuhp/0	root	0	12	N/A	N/A	N/A	N/A	N/A	0	Normal
cpuhp/1	root	0	13	N/A	N/A	N/A	N/A	N/A	0	Normal
watchdog/1	root	0	14	N/A	N/A	N/A	N/A	N/A	0	Normal
migration/1	root	0	15	N/A	N/A	N/A	N/A	N/A	0	Normal
ksoftirqd/1	root	0	16	N/A	N/A	N/A	N/A	N/A	0	Normal
kworker/1:0H	root	0	18	N/A	N/A	N/A	N/A	N/A	-20	Very High
cpuhp/2	root	0	19	N/A	N/A	N/A	N/A	N/A	0	Normal
watchdog/2	root	0	20	N/A	N/A	N/A	N/A	N/A	0	Normal
migration/2	root	0	21	N/A	N/A	N/A	N/A	N/A	0	Normal
ksoftirqd/2	root	0	22	N/A	N/A	N/A	N/A	N/A	0	Normal
kworker/2:0H	root	0	24	N/A	N/A	N/A	N/A	N/A	-20	Very High

System Boot

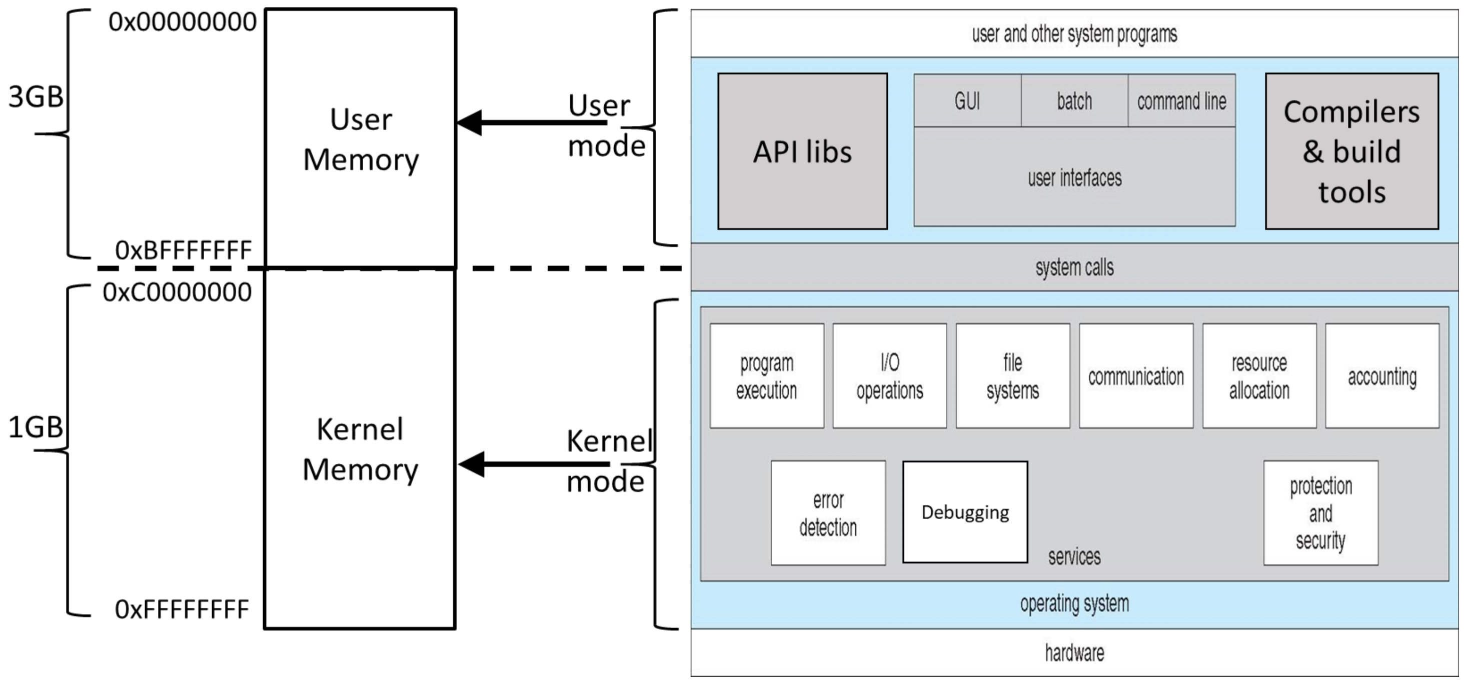
- When powering up the system, execution starts at a fixed memory location
  - Firmware ROM used to hold initial boot code
- Operating system must be made available to hardware so hardware can start it
  - Small piece of code – **bootstrap loader**, stored in **ROM** or **EEPROM** locates the kernel, loads it into memory, and starts it
  - Sometimes two-step process where **boot block** at fixed location loaded by ROM code, which loads a secondary bootstrap loader (SBL) from disk
- Common second stage boot loaders (SBL):
  - **GRUB**, allows selection of kernel from multiple disks, versions, kernel options. GRUB is popular for x86 CPUs
  - **UBoot** is the popular SBL used with ARM CPUs (common with embedded Linux).
- Kernel loads and system is then **running**

kernel space / user space – 32-bit linux example



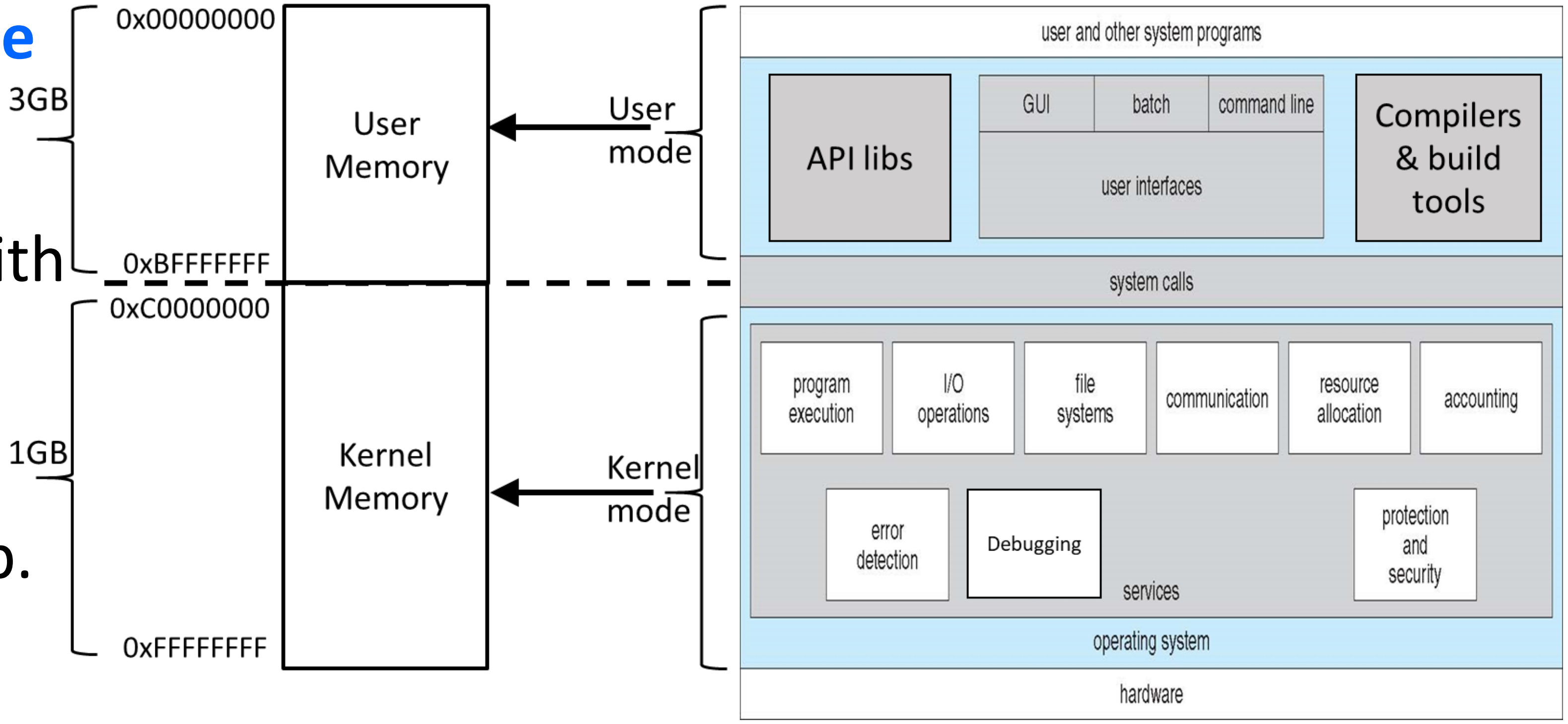
kernel space / user space – 32-bit linux example

- User-mode applications **cannot directly access** (i.e. read, write or execute) memory locations in the kernel memory. This is enforced by a piece of hardware (memory management unit) which we shall study in Lectures 6 and 7.
  - If a user program attempts to call a function in kernel memory or read a variable in kernel memory → an **illegal operation interrupt** occurs (by MMU).

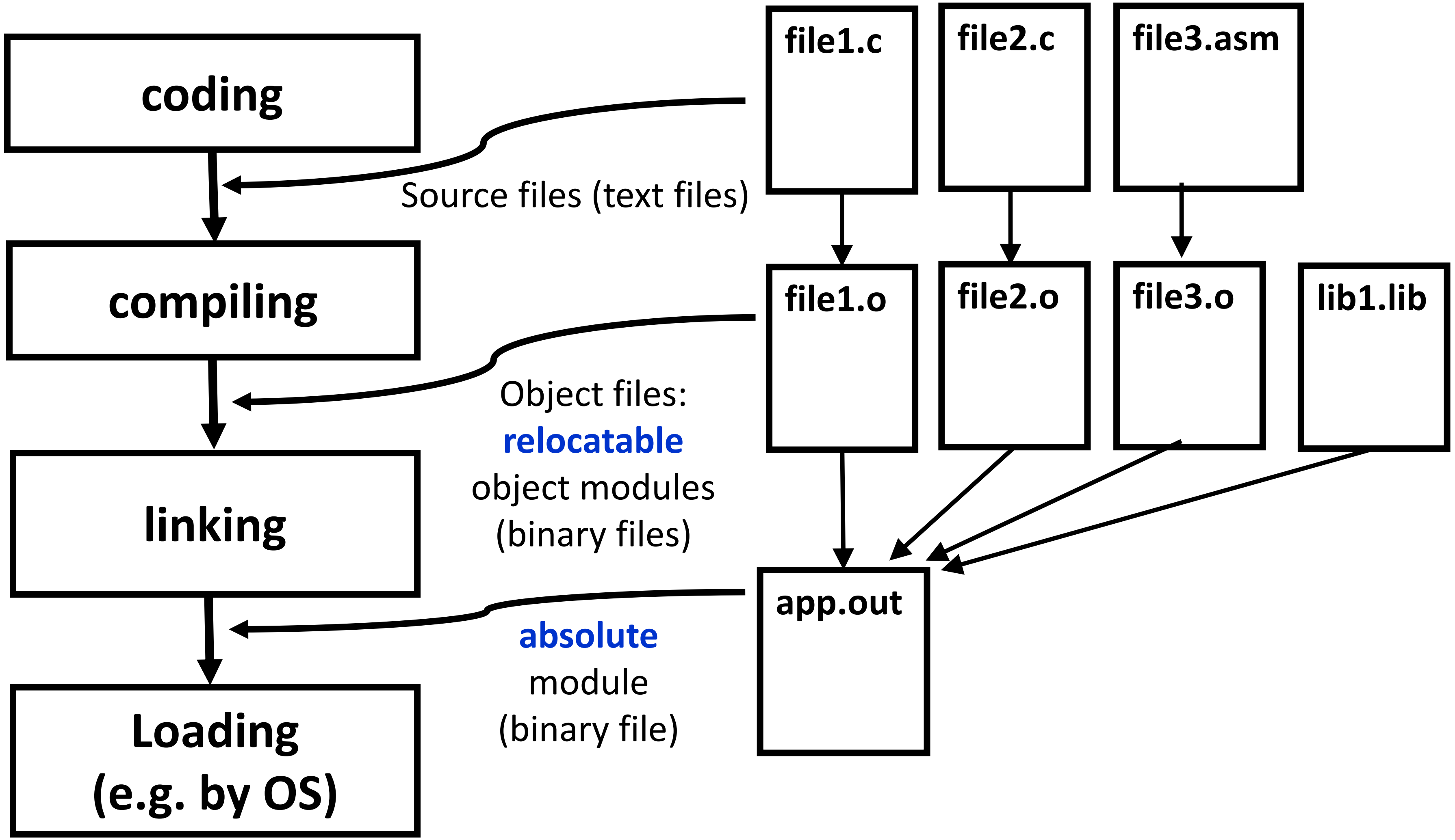


kernel space / user space – 32-bit linux example

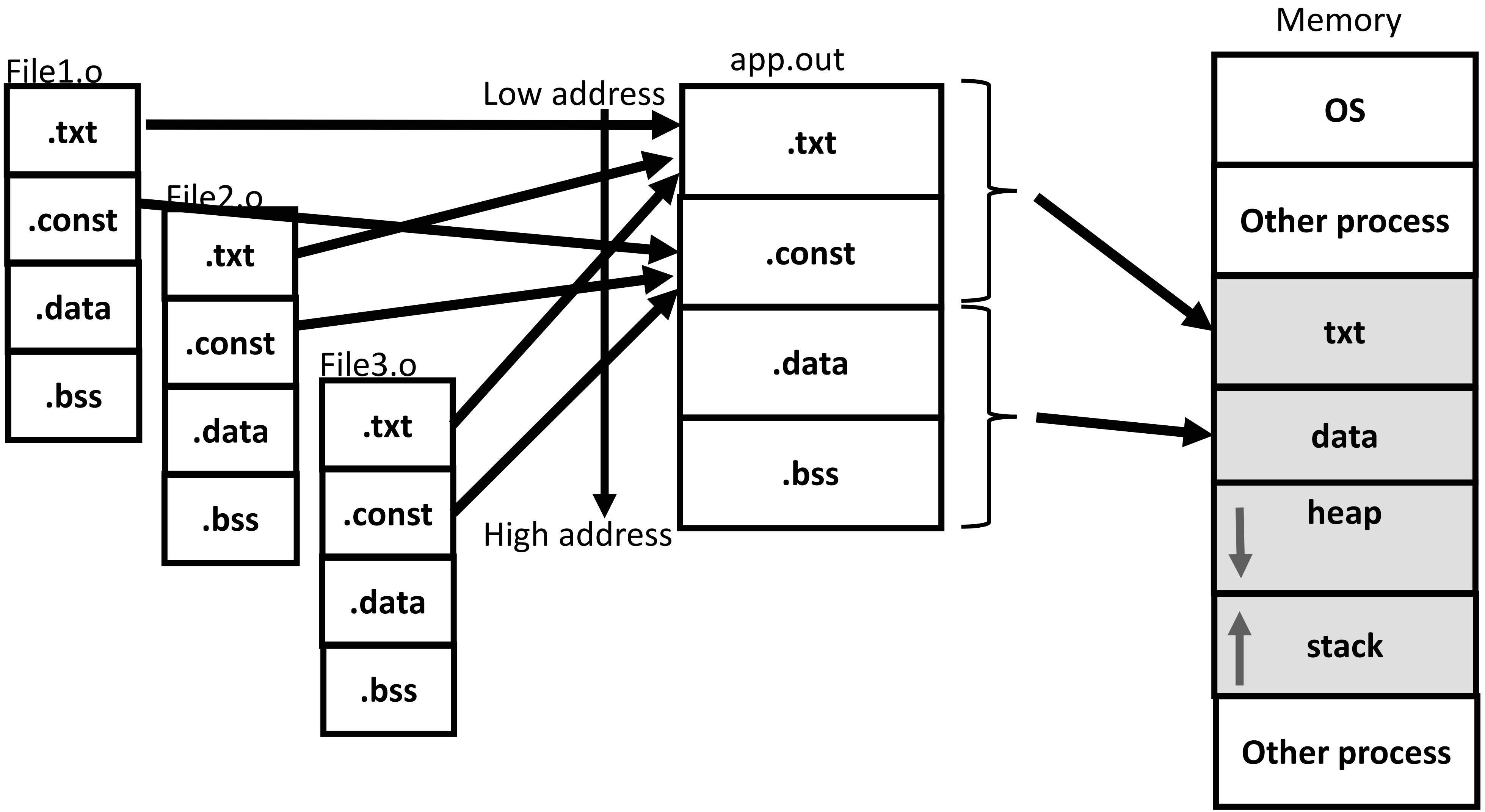
- Thus, the only way for a user program to invoke kernel functions **is via the system call interface**, in which a trap machine instruction is invoked, with the correct paramaters passed via registers/memory/stack prior to invoking the trap.

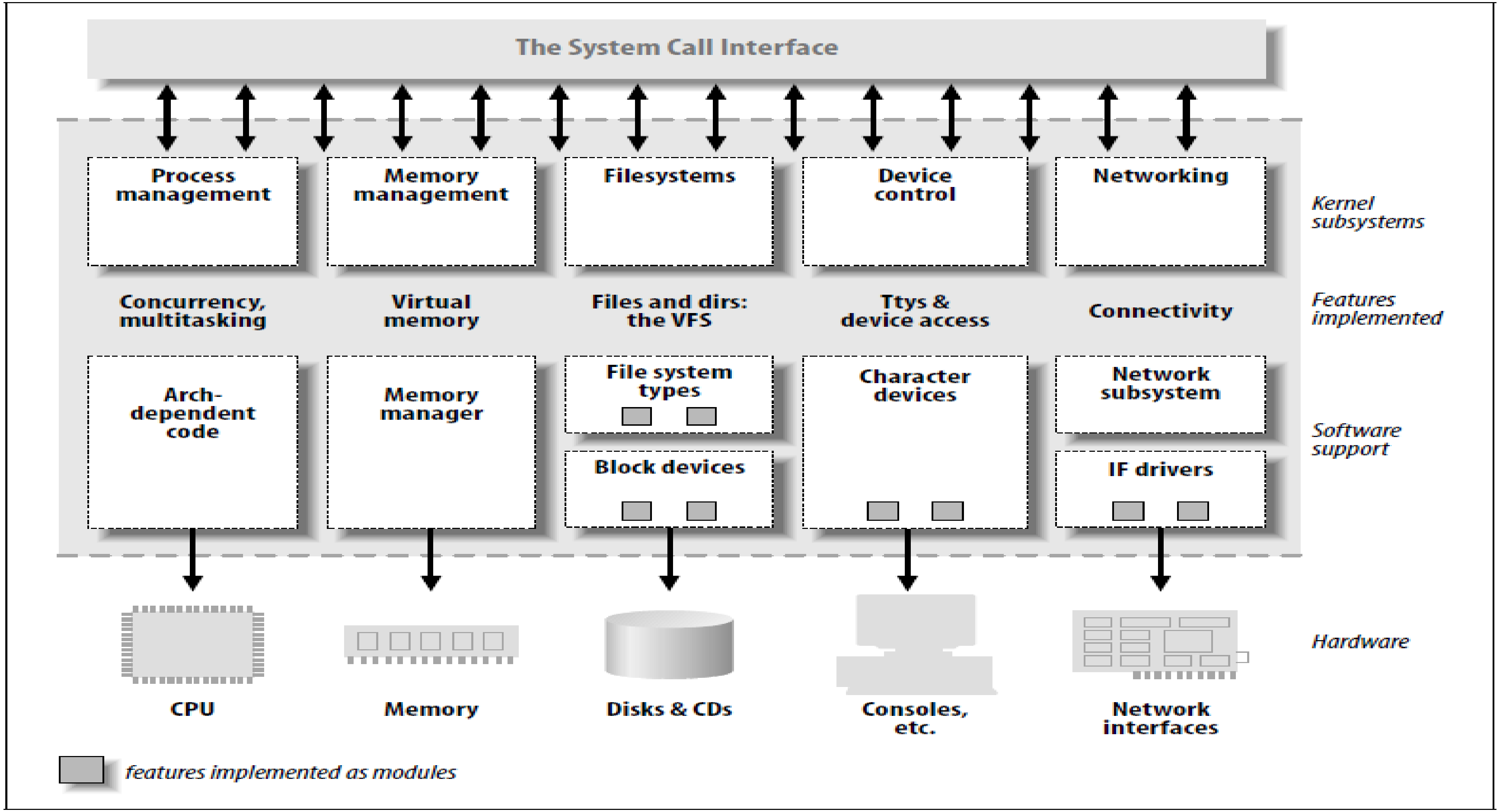


Program translation – (user-mode programs)

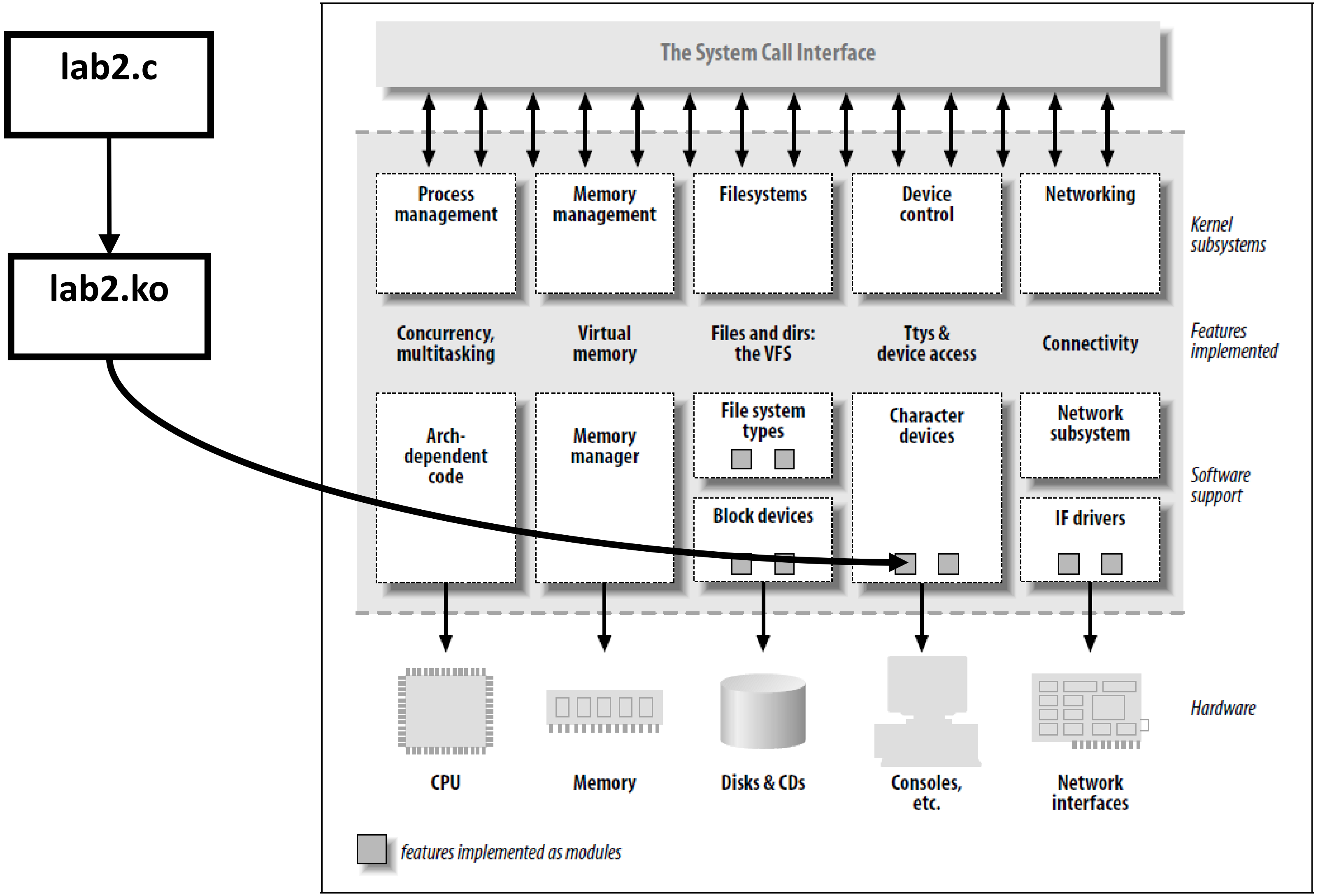


Program translation – linking + loading (user-mode programs)




The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
Hardware
features implemented as modules

Program translation – (kernel modules)



Additional resource

https://www.kernel.org/doc/html/latest/

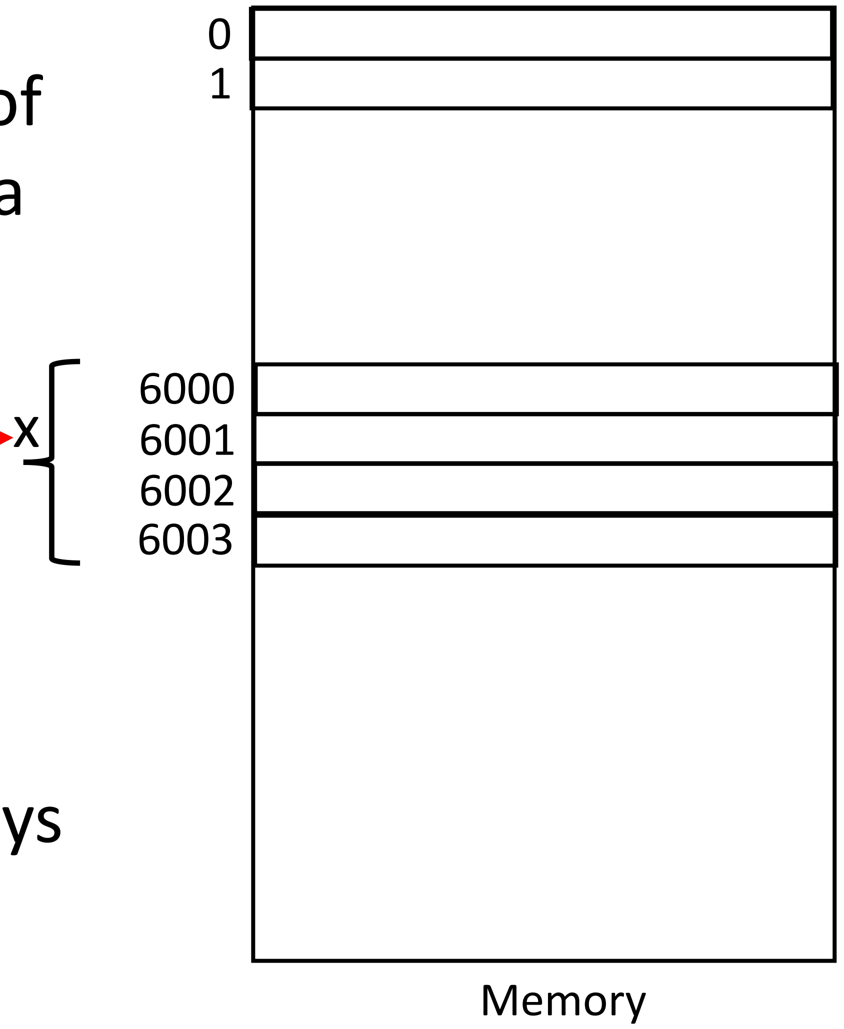
https://lwn.net/Kernel/

https://lwn.net/Kernel/LDD3/

C-language review - Pointers

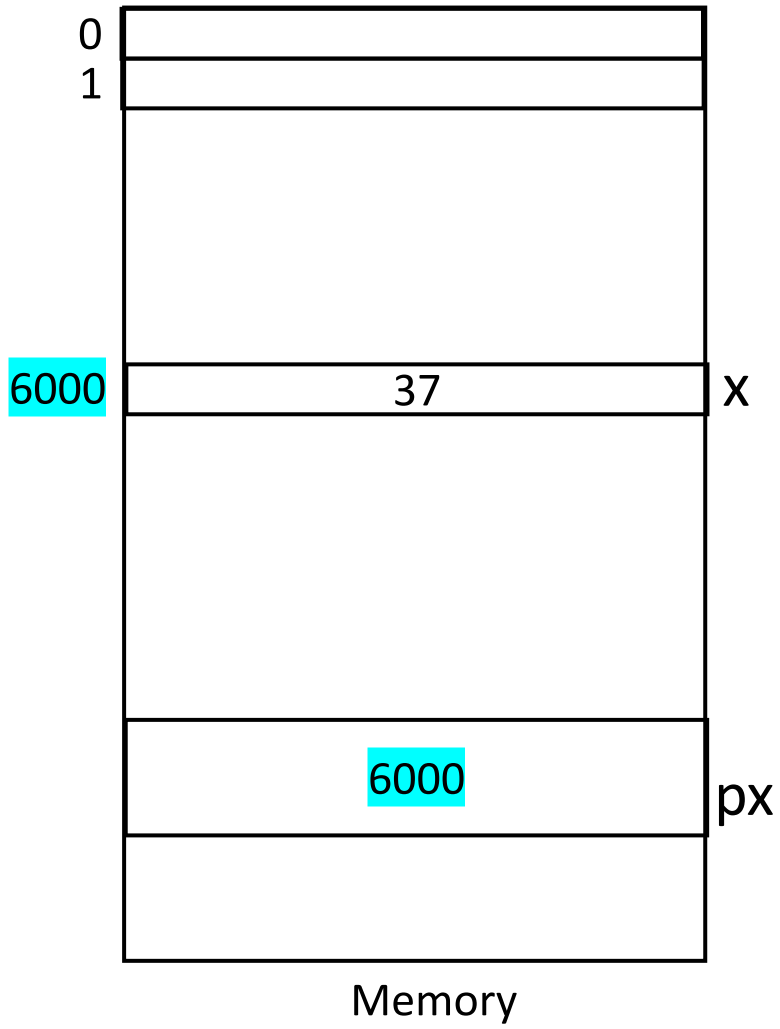
- Declaring a C variable allocates a number of memory cells (or bytes) and assigns them a name (the name of the variable).
- Memory is organized into bytes (or cells), where each has a unique address.
- When a variable is allocated a certain number of bytes in memory, they are always contiguous, for example:

```
int x;
```



Pointers (cont.)

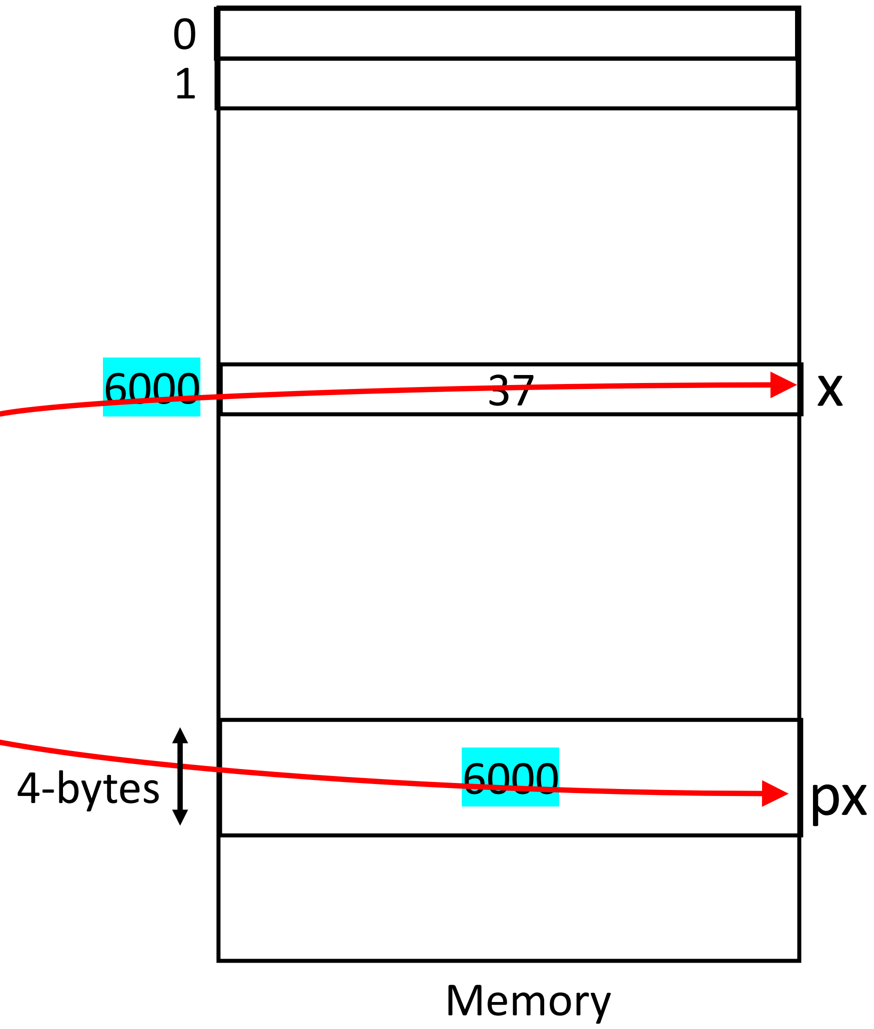
- In many cases, your program may need to know the memory address of your variable and may also need to access a variable using its address instead of its name.
  - Note: An address is always a byte-address (e.g. address 6000 tells you integer x is preceded by 6000 bytes before it, NOT 6000 integers, 4-bytes each).



Pointers – The address of operator &

- You can access the address of a variable using the & operator, e.g.

```
char x = 37;
char *px;
px = &x
```



In the above example:

- & is the "**address-of**" operator
- **`char*` is the declaration** of a pointer

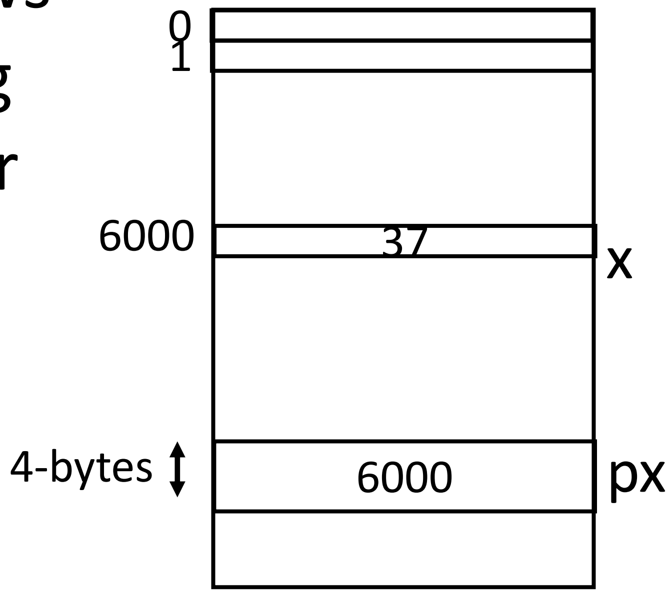
Pointers – The dereference operator *

The **dereference operator *** allows reads or writes to a variable using its pointer instead of its name, for example:

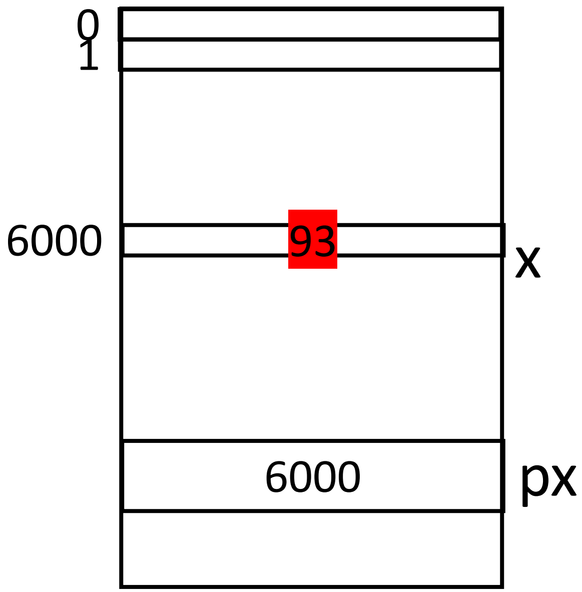
```
*px = 93;
```

change the contents of variable x from 37 to 93. Hence it is equivalent to the statement:

```
x=93;
```



Memory before the statement *px=93;



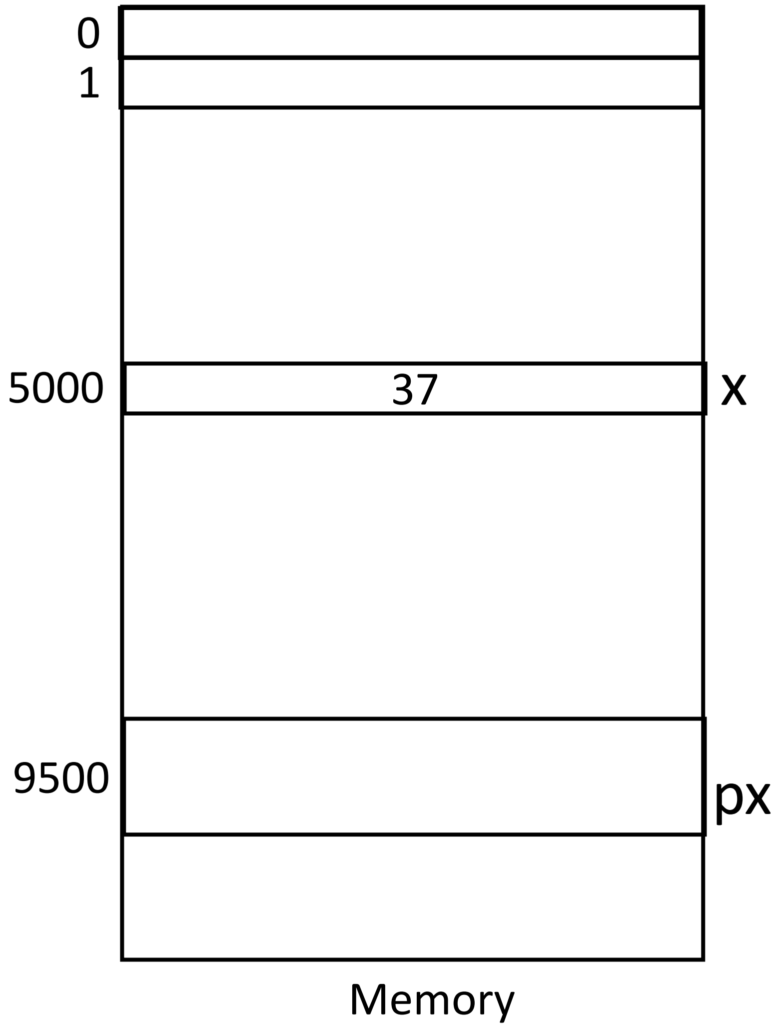
Memory after the statement *px=93;

Pointers – cont.

```
int x = 37;
int *px = &x;
```

Which one of the following statements evaluates to true?

```
(x==5000)
(x==37)
(&x==9500)
(&x==5000)
(px==9500)
(px==37)
(*px==9500)
(*px==37)
```

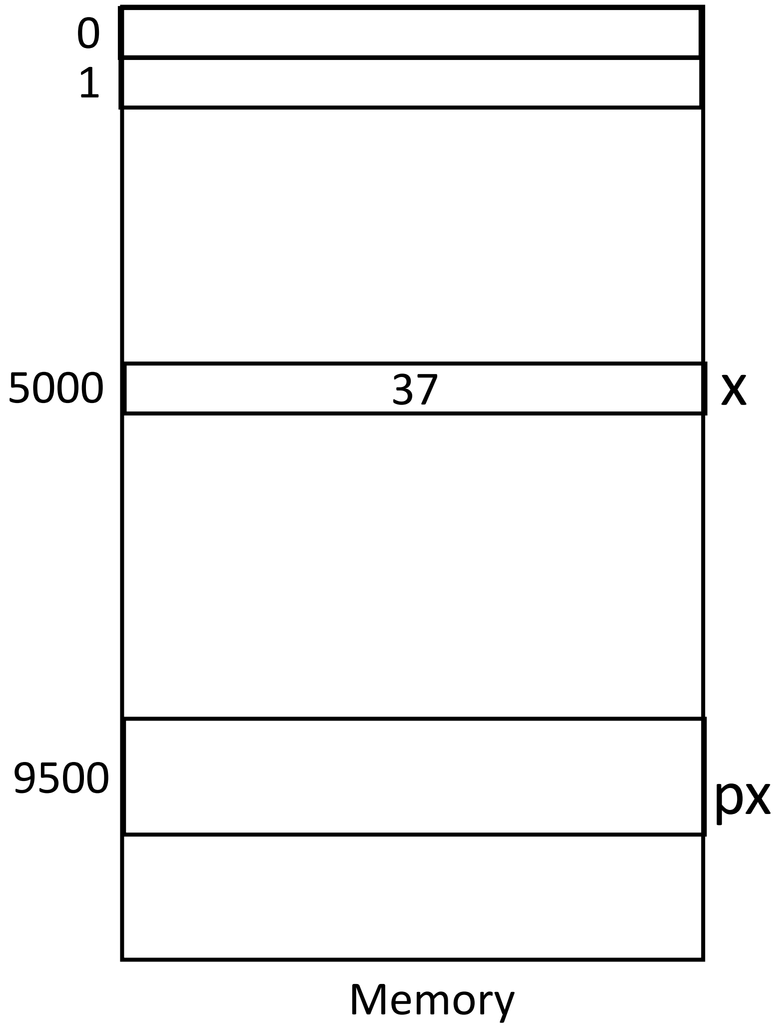


Pointers - Example:

```
int x = 37;
int *px = &x;
```

Which one of the following statements evaluates to true?

`(x==5000)`	false
`(x==37)`	true
`(&x==9500)`	false
`(&x==5000)`	true
`(px==9500)`	false
`(px==37)`	false
`(*px==9500)`	false
`(*px==37)`	true



Pointers - declaration

```
int *p1;
char *p2;
double *p3;
```

- Note that the asterisk (*) used when declaring a pointer should not be confused with the dereference operator seen earlier. They are two different things represented with the same sign.

```
int * x,y;
```

- In the previous line, `x` is declared as a pointer, but `y` is declared as an `int`.

Pointers – example 1

```
// my first pointer
#include <stdio.h>

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;

  printf("firstvalue is %d\n", firstvalue);
  printf("secondvalue is %d\n",secondvalue);
  return 0;
}
```

Pointers – example 1

```
// my first pointer
#include <stdio.h>

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;

  printf("firstvalue is %d\n", firstvalue);
  printf("secondvalue is %d\n",secondvalue);
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

Pointers – example 2

```
// more pointers
#include <stdio.h>

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 =
                     //value pointed to by p1
  p1 = p2;           // (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20

  printf("firstvalue is %d\n“,firstvalue);
  printf(“secondvalue is %d\n“,secondvalue);
  return 0;
}
```

Pointers – example 2

```
// more pointers
#include <stdio.h>

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed to by p1 = 10
  *p2 = *p1;         // value pointed to by p2 =
                     //value pointed to by p1
  p1 = p2;           // (value of pointer is copied)
  *p1 = 20;          // value pointed to by p1 = 20

  printf("firstvalue is %d\n",firstvalue);
  printf("secondvalue is %d\n",secondvalue);
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

Arrays and pointers

- The array name holds the starting address of the array

```
int vals[] = {4, 7, 11};
```

4	7	11

starting address of `vals`: `0x4a00`

```
printf("%lx", (unsigned long) vals);
            // displays 0x4a00
printf("%lx", (unsigned long) vals[0]);
            // displays 0x4
```

Arrays and pointers – cont.

- Array name can be used as a pointer (a **constant pointer**):

```
int vals[] = {4, 7, 11};
printf(“%d”, *vals);    // displays 4
```

- Pointer can be used as an array name:

```
int *valptr = vals;
printf(“%d”,valptr[1]; // displays 7
```

Arrays and pointers

- Hence, arrays work very much like pointers to their first element, and an array can always be implicitly converted to a pointer of the proper type, i.e. a ***pointer can be assigned any value, whereas an array can only represent the same elements it pointed to during its instantiation***, hence:

```
int x[20];
int *px;

px = x;
```

valid

```
x = px;
```

Not valid

- An array declaration allocates memory for the number of elements inside the array, whereas the declaration of a pointer allocates only the memory required to hold an address.

Arrays and pointers – example

```
// more pointers
#include <stdio.h>

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    printf("%d, ", numbers[n]);
  return 0;
}
```

Array and pointers – example

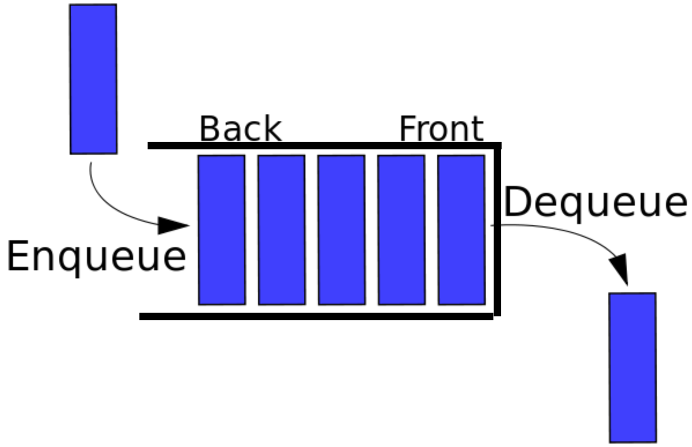
```
// more pointers
#include <stdio.h>

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    printf("%d, ", numbers[n]);
  return 0;
}
```

```
10, 20, 30, 40, 50,
```

The Queue data structure: (first in, first out – FIFO)

- Queue: a FIFO (first in, first out) data structure.
- Examples:
  - people in line at the theatre box office
  - print jobs sent to a printer
  - Input from a keyboard is buffered into a stream using a fixed size FIFO.
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list



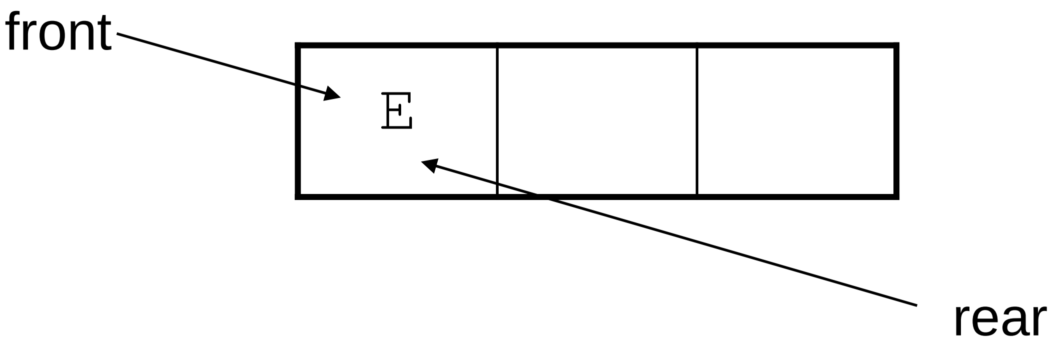
The Queue data structure - operations

- Locations
  - Back/Rear (tail): position where elements are added
  - Front (head): position from which elements are removed
- Operations:
  - enqueue: add an element to the rear of the queue
  - dequeue: remove an element from the front of a queue

Queue operations – cont.

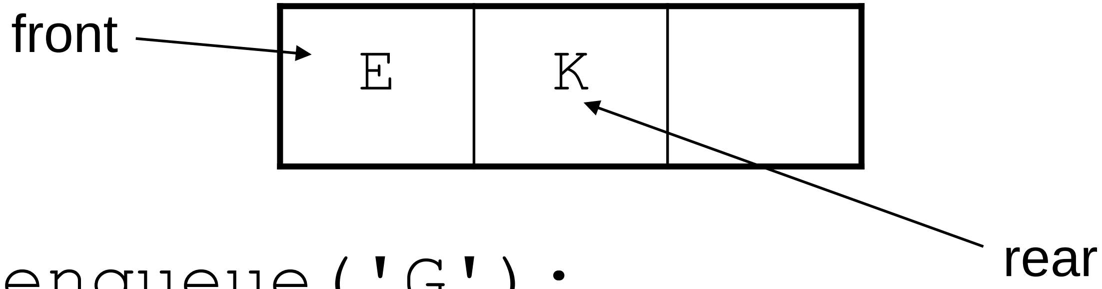
- A currently empty queue that can hold `char` values:

- `enqueue('E');`

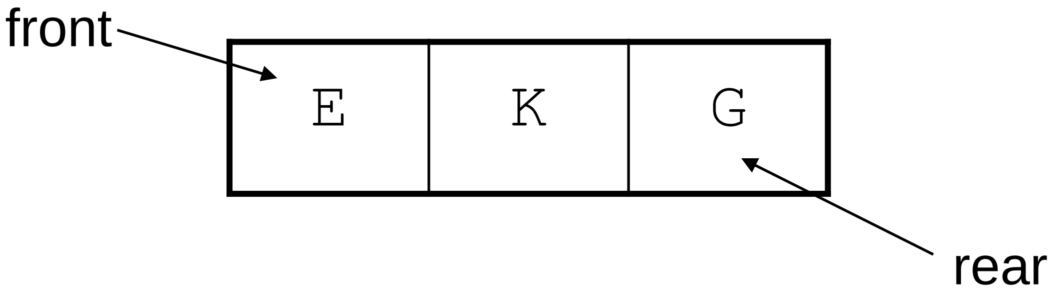


Queue operations – cont.

- `enqueue('K');`

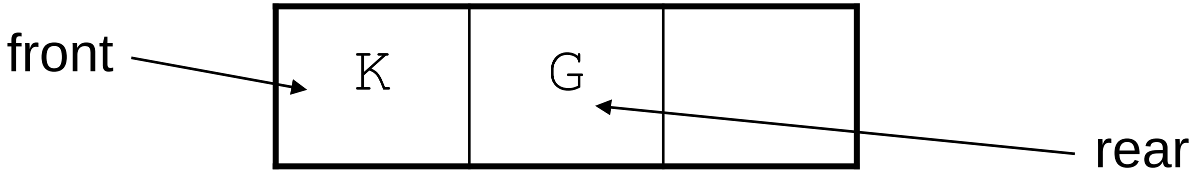


- `enqueue('G');`

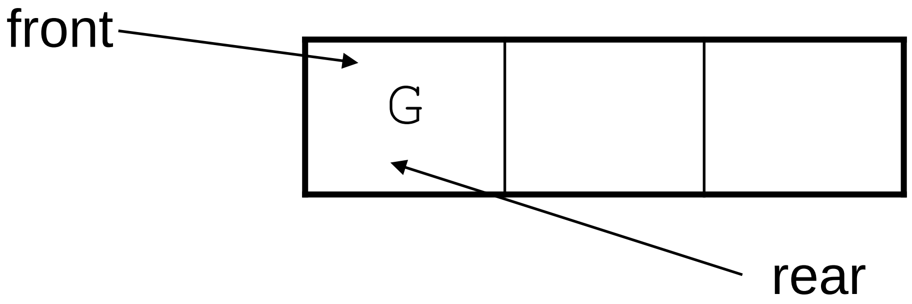


Queue operations – cont.

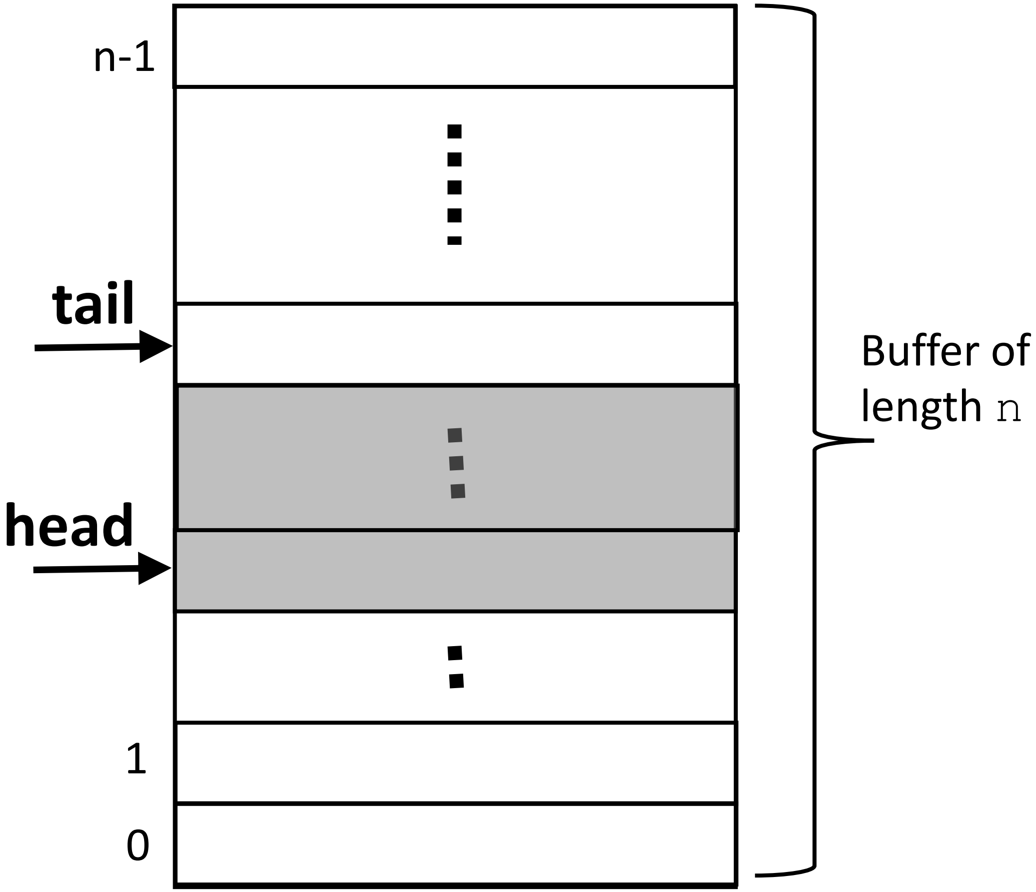
- `dequeue(); // remove E`

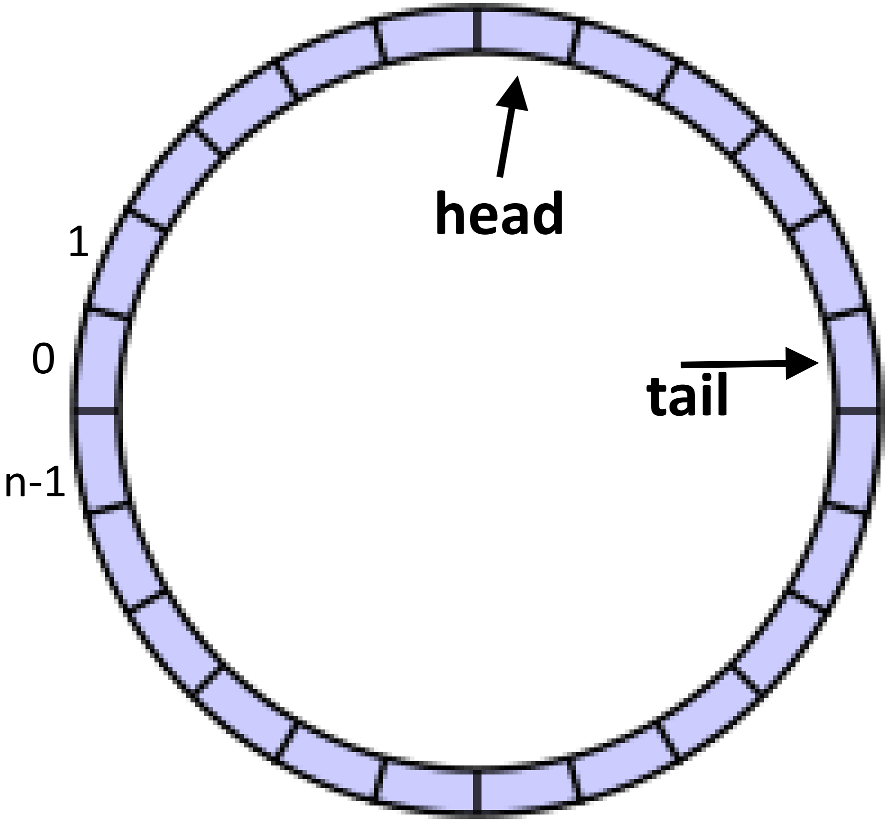


- `dequeue(); // remove K`



The Queue data structure – Buffer Implementations





The queue data structure - Design/algorithms

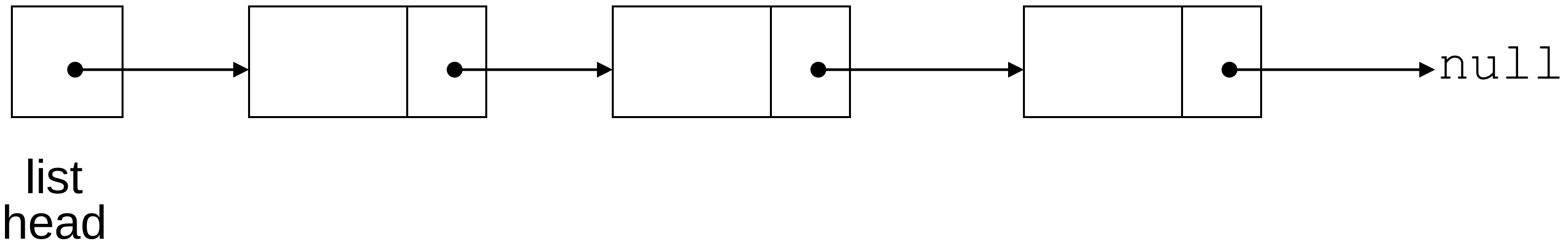
- Create a buffer of length n
- Create some variables : (all initialized to zero)
  - Use a variable (tail or write_index) to hold the index where new data should be written.
  - Use a variable (head or read_index) to point to the array index from which data may be read.
  - A count variable to hold the current number of elements in the array.
- Before enqueuing, make sure that counter < n (i.e. there is room for adding an entry). If not, return a failure.
- To enqueue an entry, write it to the array using the tail variable and then increment the tail using modulo n arithmetic. Also increment the counter

The queue data structure - Design/algorithms

- Before dequeuing, make sure that counter > 0, else return a failure.
- To dequeue an entry, read it out using the head variable, increment the head using modulo n arithmetic and also decrement the counter.

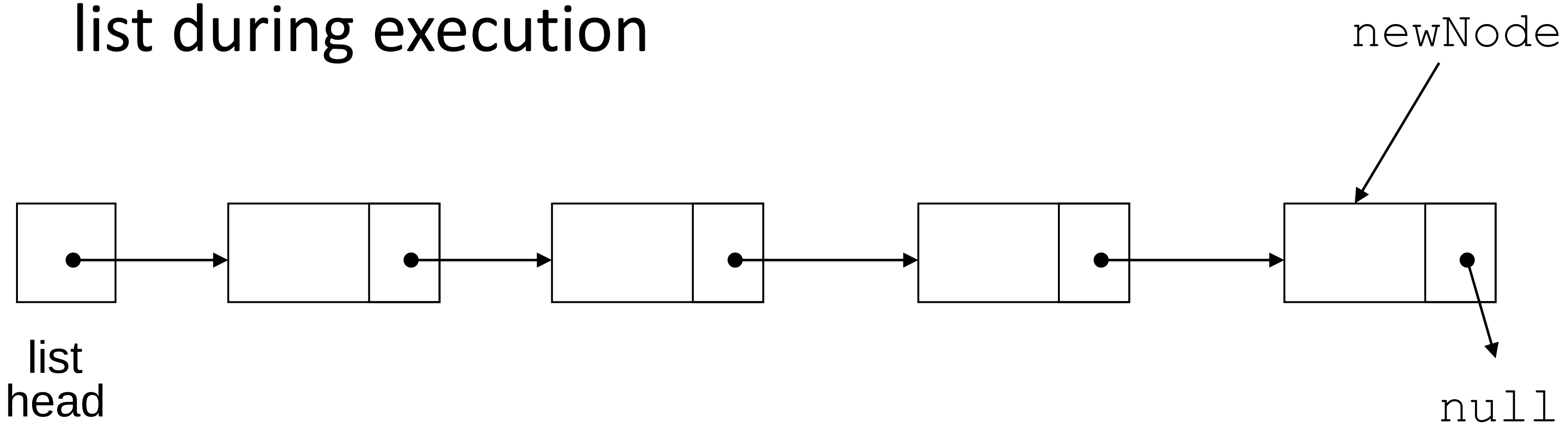
Introduction to the Linked Lists

- <u>Linked list</u>: set of data structures (<u>nodes</u>) that contain references to other data structures



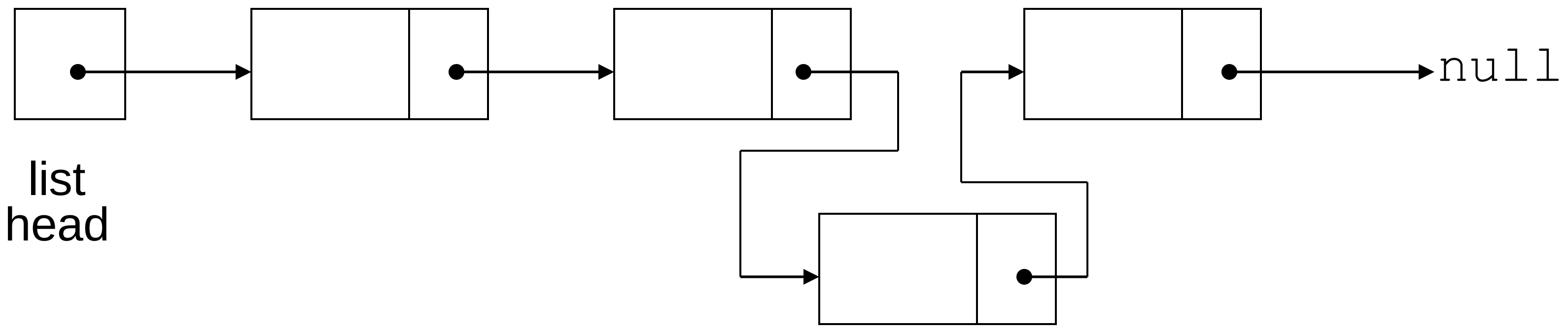
Introduction to the Linked Lists - cont.

- References may be addresses or array indices. In our OS class, the kernel uses addresses to point to the next node in the list.
- Nodes may be added to or removed from the linked list during execution



Linked Lists vs. Arrays

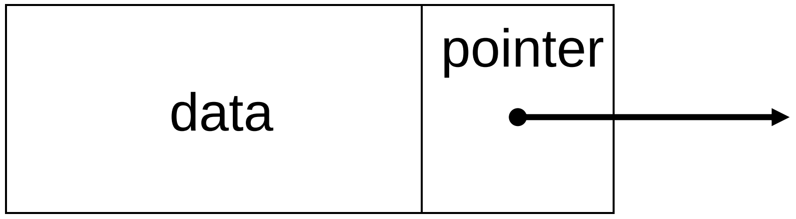
- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size
- Linked lists can insert a node between other nodes easily



Node Organization

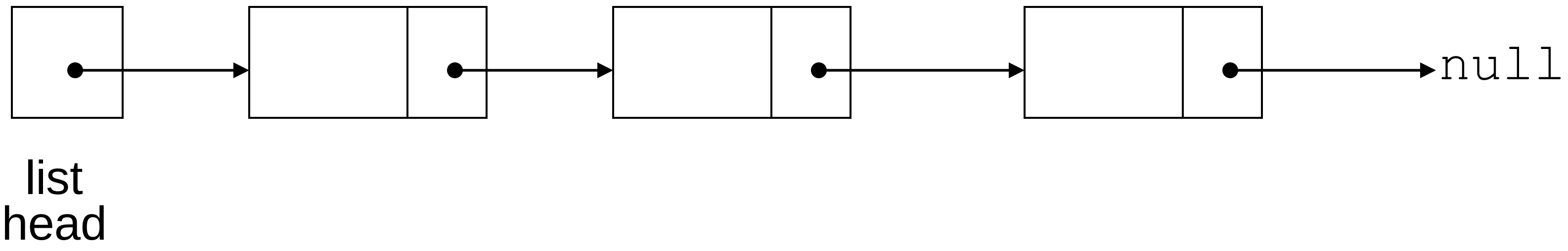
- A node contains:
  - **data**: one or more data fields – may be organized as structure, object, etc.
  - **a pointer**: that can point to another node

- In our case, the data is the data in the process control block (PCB), whereas the pointer is the next PCB (for the next process)



Linked List Organization

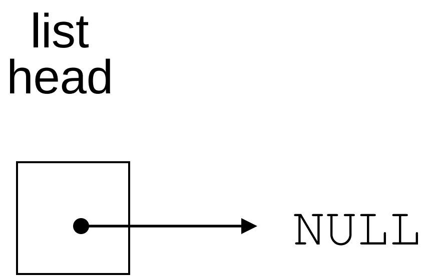
- Linked list contains 0 or more nodes:



- Has a list head to point to first node
- Last node points to `null` `(address 0)`

Empty List

- If a list currently contains 0 nodes, it is the <u>empty list</u>
- In this case the list head points to `null`



Declaring a Node

- Declare a node:

```
struct ListNode
{
        int data;
        ListNode *next;
};
```

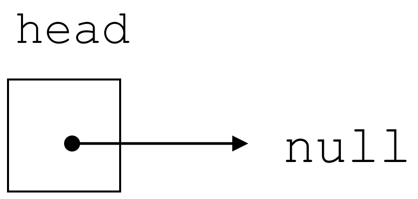
- No memory is allocated at this time

Defining a Linked List

- Define a pointer for the head of the list:

```
ListNode *head = nullptr;
```

- Head pointer initialized to `nullptr` to indicate an empty list
- A queue (= FIFO) may be implemented as a linked list.
  - The head of the FIFO is the first entry in the list. Tail is the last entry.



The Null Pointer

- Is used to indicate end-of-list (null = address 0 in memory which normally not a valid address)
- Should always be tested for before using a pointer:

Linked List Operations

- Basic operations:
  - append a node to the end of the list
  - insert a node within the list
  - traverse the linked list
  - delete a node
  - delete/destroy the list

Linked List Queue Operations

- Basic operations:
  - Remove a node from the head
  - Add a node to the tail
  - Queue head and Queue tail are pointers to head node and tail node respectively