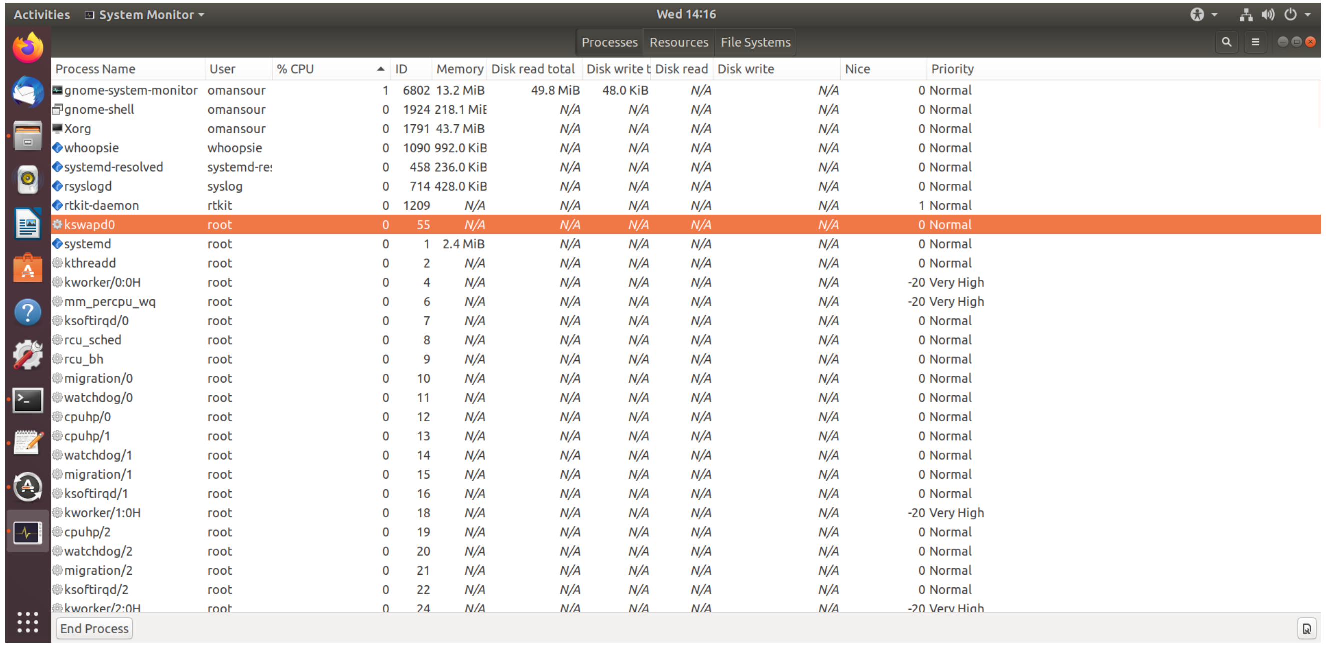
Operating-System Debugging

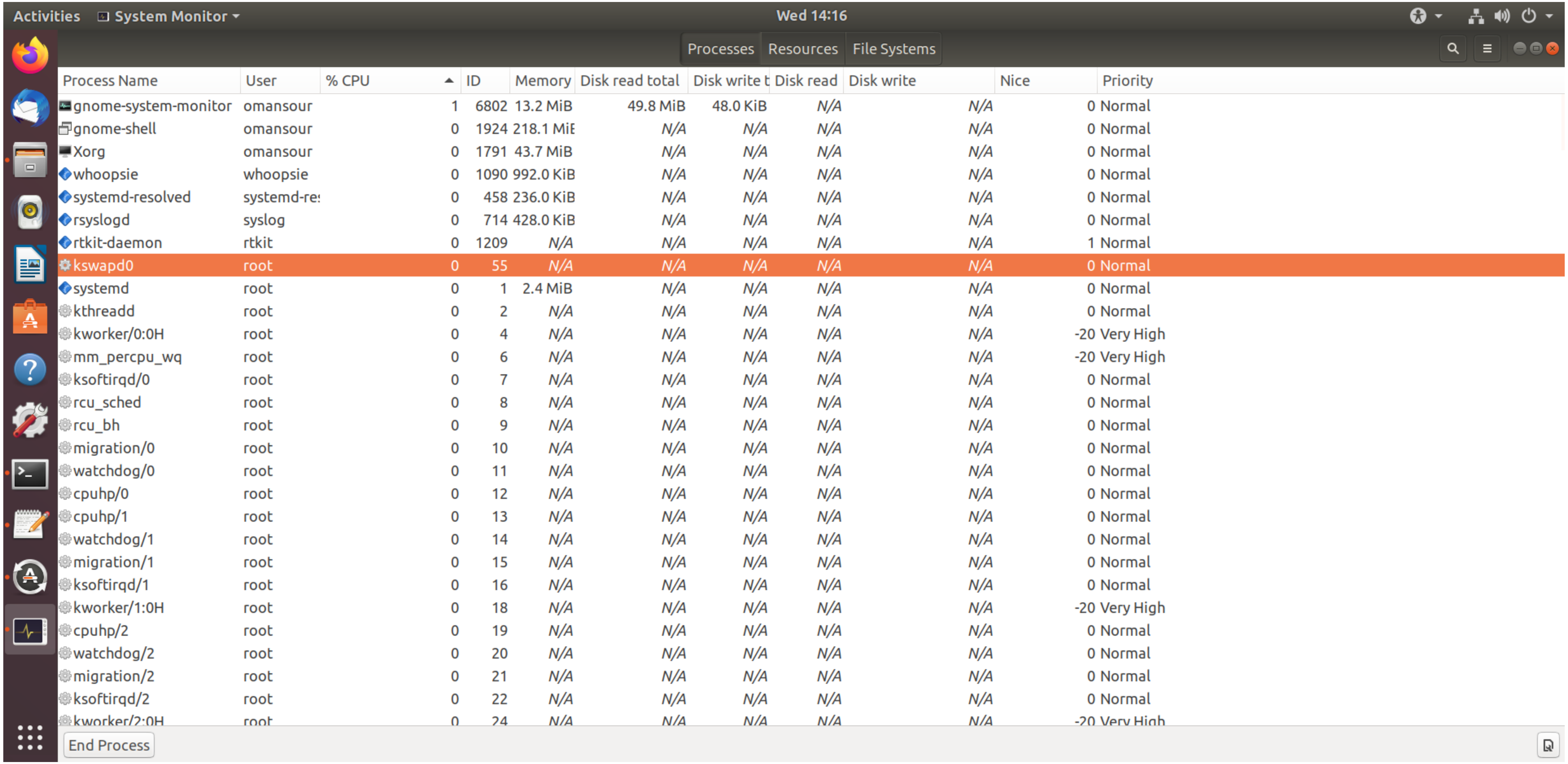
- **Debugging** is finding and fixing errors, or **bugs**
- OS generate **log files** containing error information
- Failure of an application can generate **crash dump** file capturing memory of the process.
  - Location may be found or modified by editing the file /proc/sys/kernel/**core_pattern**
- Operating system failure can generate **crash dump** file containing kernel memory
  - /var/crash/**vmcore**, but configured in /etc/kdump.conf
- 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



- 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**"

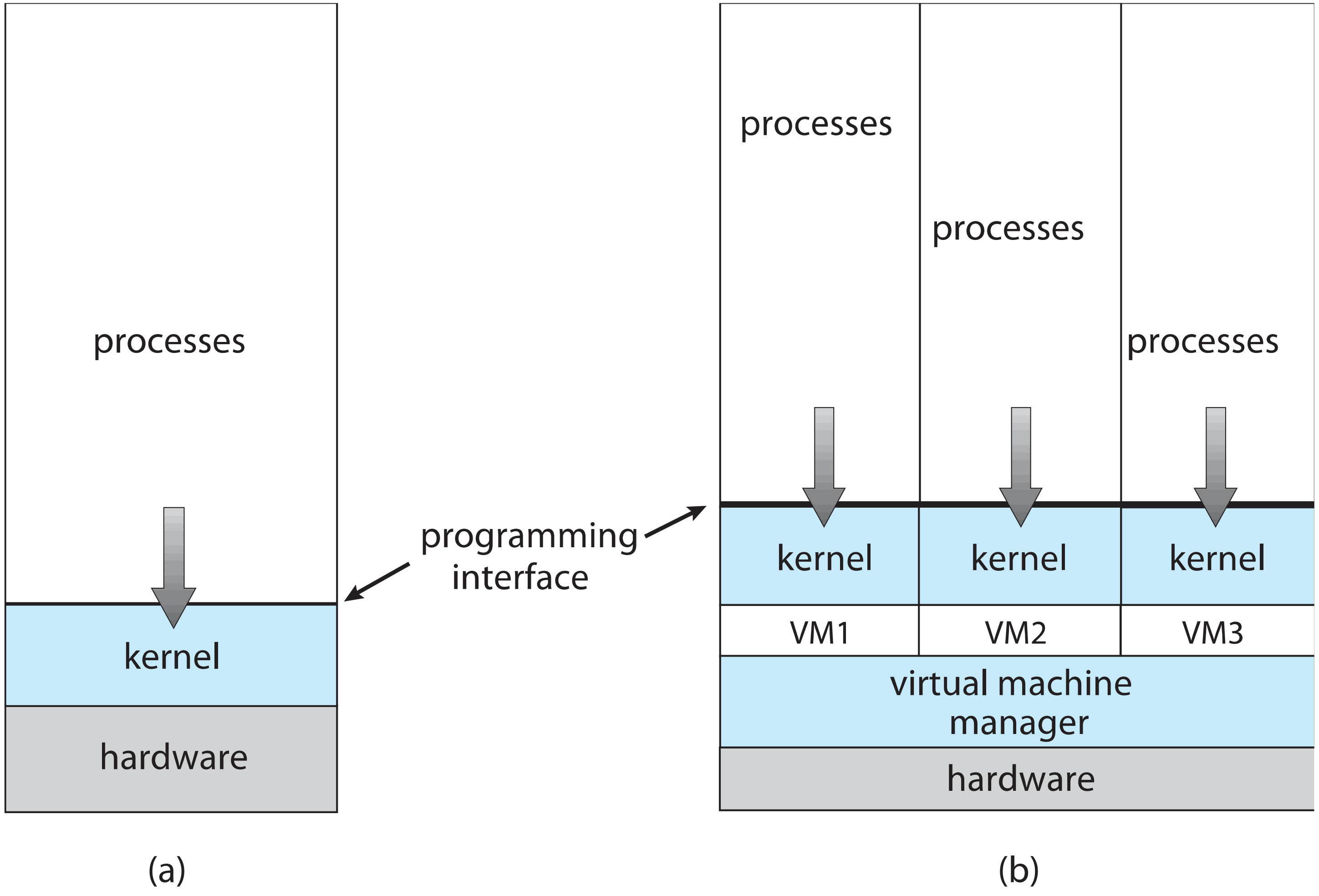
Performance Tuning



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

Virtualization

- Allows operating systems to run applications within other Oses
  - Vast and growing industry
- A **host** system with an OS and a VMM may run one or more **guest** systems, each with a different OS
- Some VMM's may install directly on Hardware, without a need for a host OS.



Virtualization – cont.

- **Emulation** used when guest system is compiled for a CPU type that is **different** from that of the host.
  - **Interpretation** is used to map guest machine instructions to host CPU's machine instructions. E.g. Apple's Parallels
  - Generally slow.
  - e.g. old Apple software (PowerPC CPUs) running on newer Apple machines (x86 CPUs).
  - e.g. Running a windows system on Apple machines prior to 2006 (PowerPC CPU's) or on Apple machines running M1/M2 CPU's.

- **Virtualization** – Host OS natively compiled for CPU architecture (always the case), running **guest** OSes also natively compiled for same CPU instruction set architecture.
  - **VMM** (virtual machine Manager) provides virtualization services
  - Consider a Windows 10 **host** OS running running Windows 10 guests (via Vmware VMM)
  - Or, a Windows 10 host running Ubuntu Linux (via Vmware or VirtualBox VMM)

Virtualization - cont.

- **Use cases** involve laptops, desktops or servers running multiple OSes for **exploration or compatibility**
  - Apple laptop running Mac OS X host, Windows as a guest: e.g. parallels "Desktop" and VMware's "Fusion" allows a Mac system to run windows and windows applications, or Linux.
  - Windows laptop with Linux as a guest: We shall use this setup for our labs, using VMware workstation.
  - **Developing apps** for multiple OSes without having multiple systems
  - **QA testing** applications without having multiple systems
  - Executing and managing compute environments **within data centers**
- A VMM may also run without a host OS (Type 2 Hypervisor)
  - VMM replaces the general-purpose OS and becomes the host OS.
  - e.g. VMware ESX and Citrix XenServer

Open-Source Operating Systems

- Operating systems made available in source-code format rather than just binary **closed-source**
- Examples include:
  - **GNU-Linux** and
  - **Android-Linux**
  - **BSD UNIX** (e.g. FreeBSD and NetBSD)
  - and a few more.
- When building or developing applications and drivers for an open source OS, good attention must be paid to the licensing model:
  - Reciprocal: e.g. GPLv1, GPLv2, etc.
  - Permissive: e.g. BSD, MIT, etc.

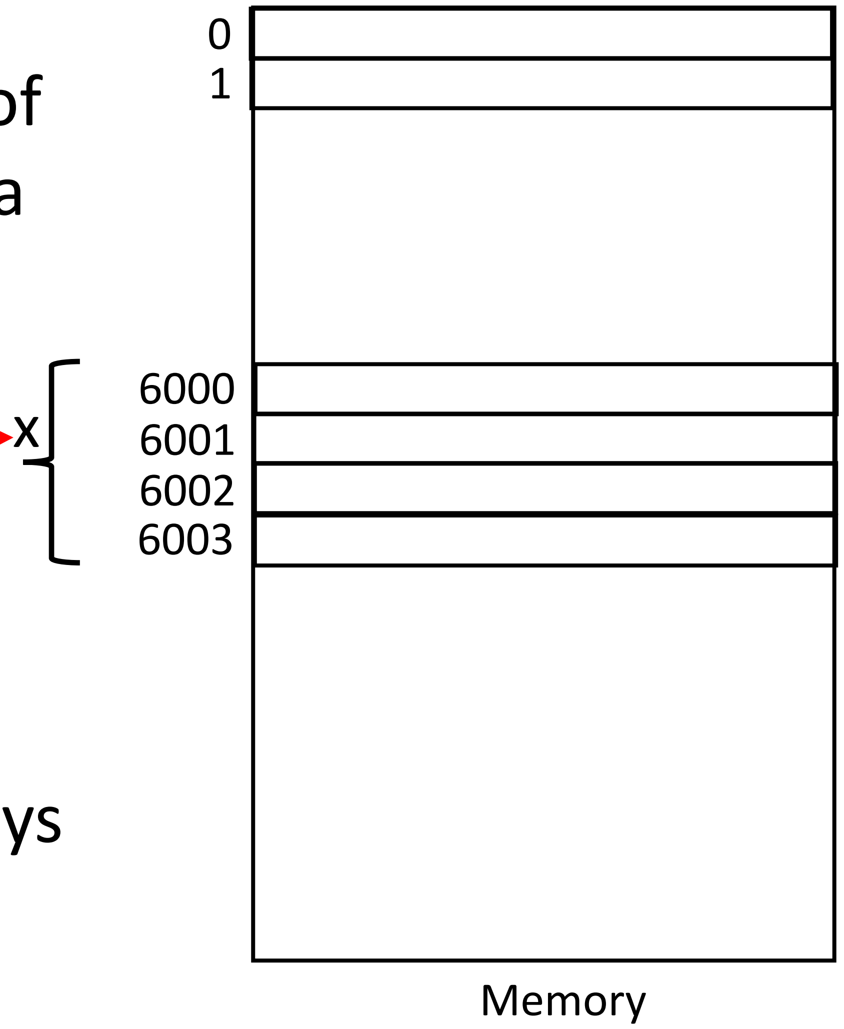
Review

- C – review
  - Function pointers
- Linking and kernel modules
- Elementary data structures - review

C-language review - Pointers

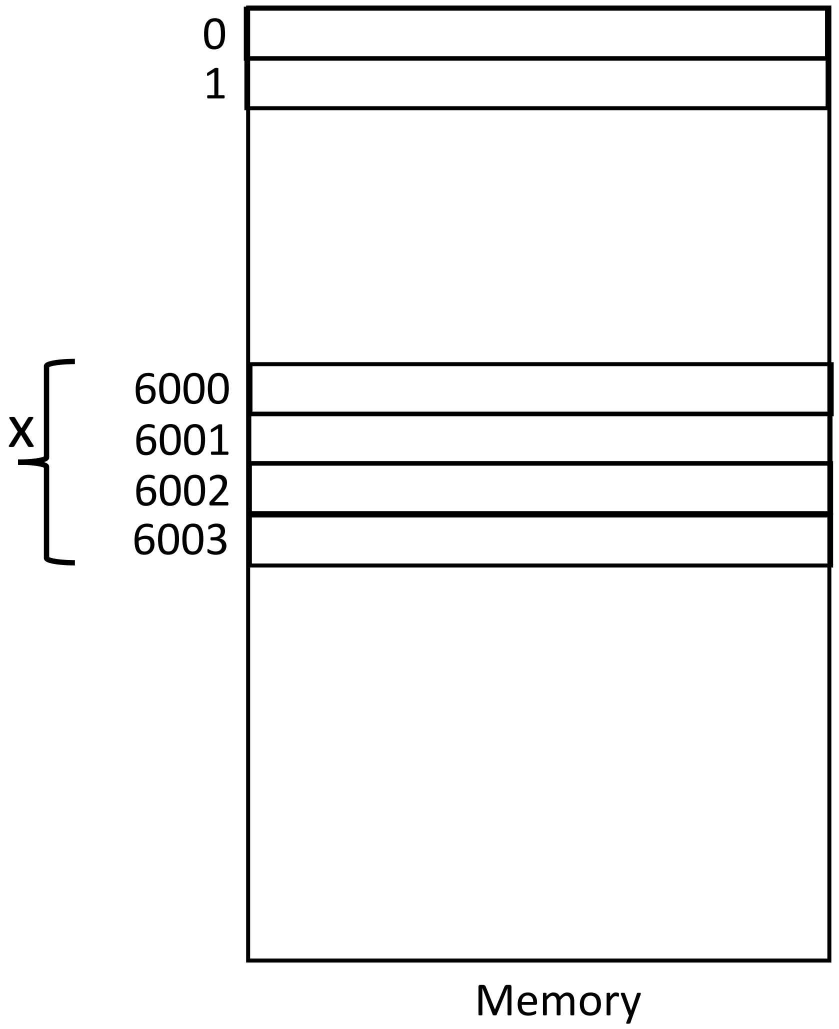
- Declaring a C variable allocates a number of memory cells (or bytes) and assigns them a name (the name of the variable).
- Memory is organized into bytes (or cells), where each has a unique address.
- When a variable is allocated a certain number of bytes in memory, they are always contiguous, for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

```
int x;
```



Pointers (cont.)

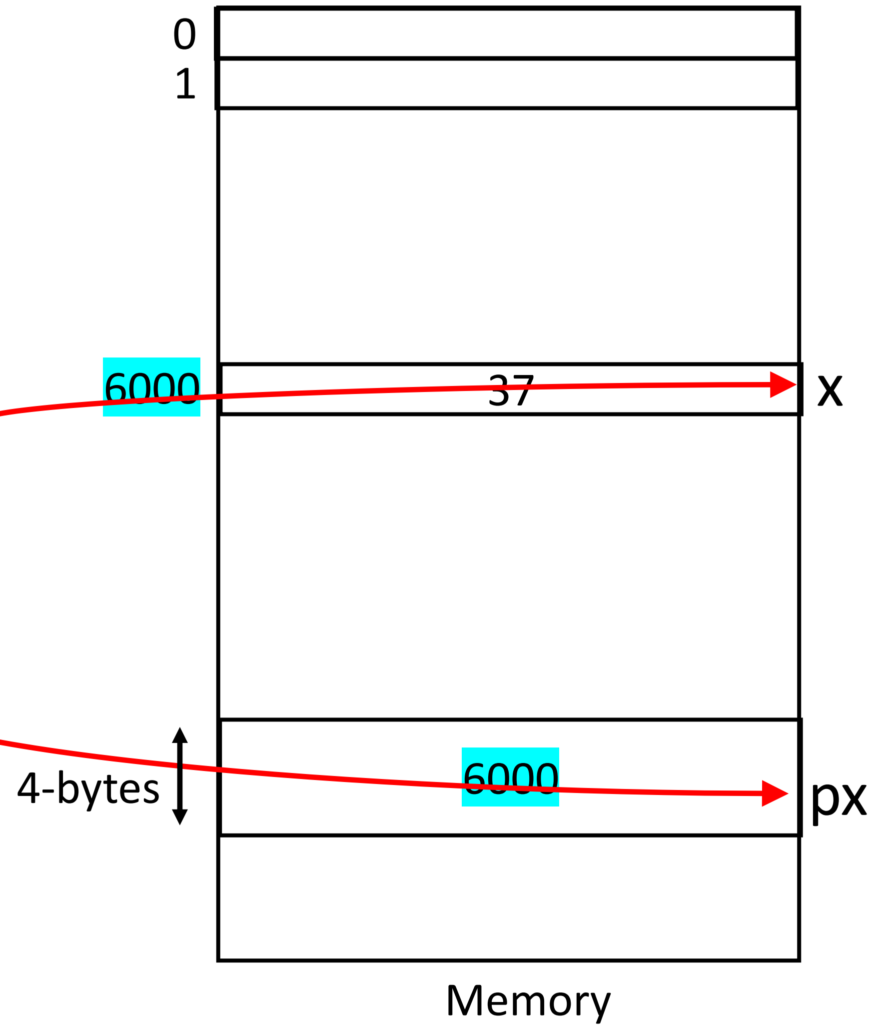
- In many cases, your program may need to know the memory address of your variable and may also need to access a variable using its address instead of its name.
    - Note: An address is always a byte-address (e.g. address 6000 tells you integer x is preceded by 6000 bytes before it, NOT 6000 integers,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



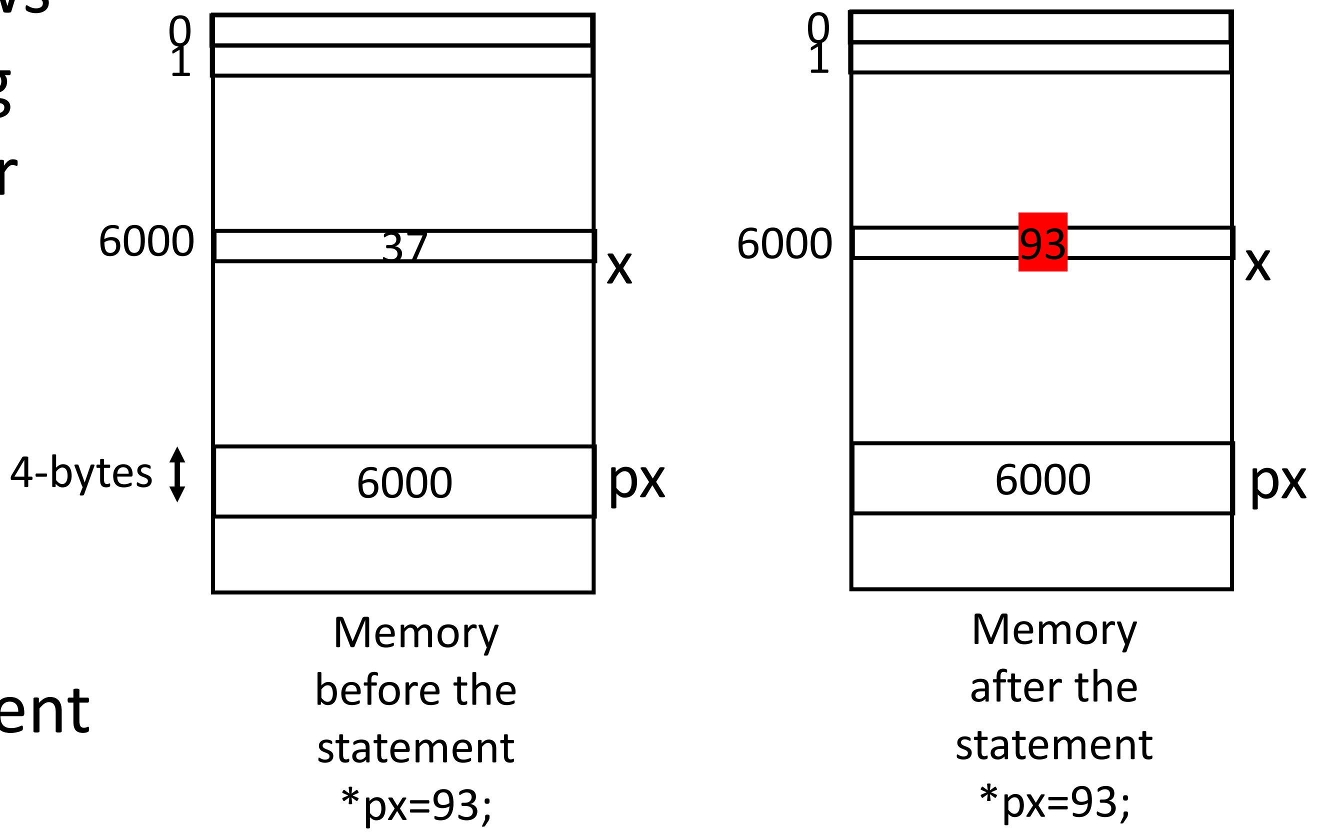
In the above example:

- `&` 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:

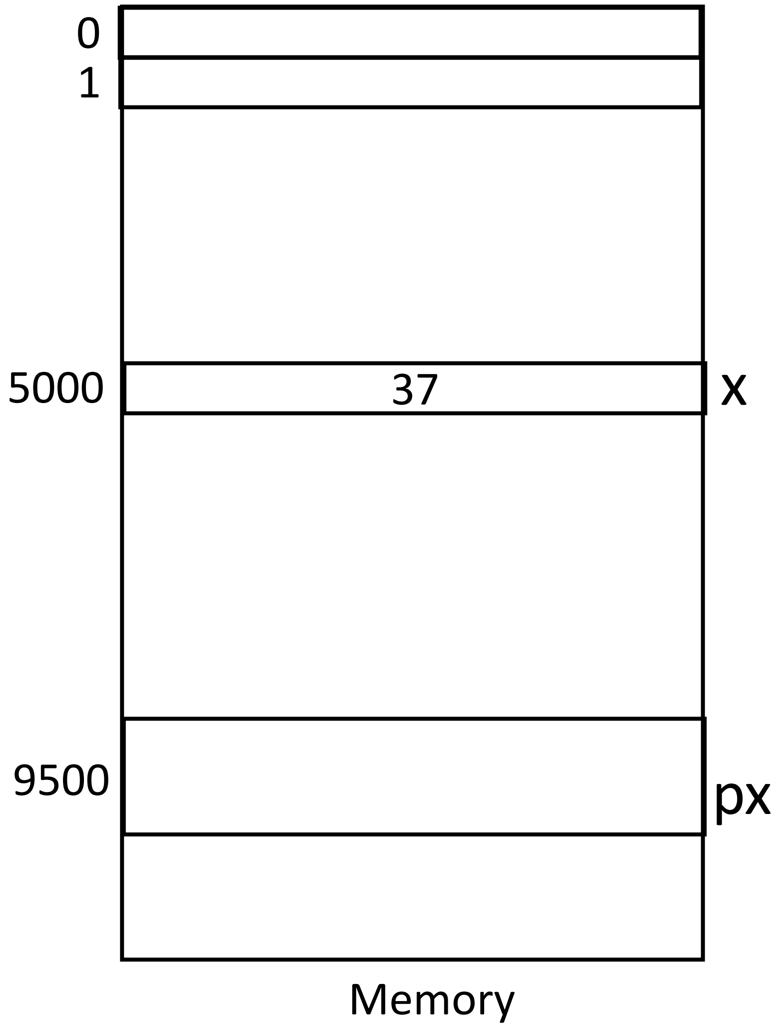
```
x=93;
```

Pointers – cont.

```
char x = 37;
char *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)
```

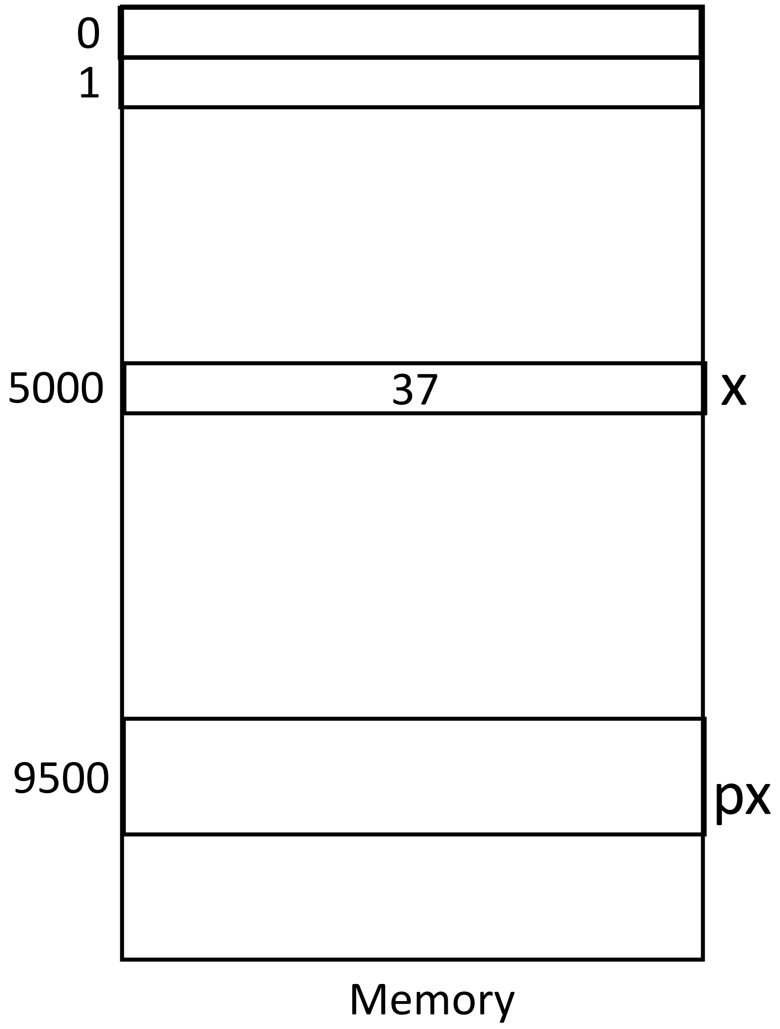


Pointers - Example:

```
char x = 37;
char *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
printf("%d",*valptr); // displays ??
printf("%d",valptr[0]); // displays ??
printf("%d",*(valptr+1)); // displays ??
```

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;
```

valid

Not valid

```
px = x;                    x = px;
```

- 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,
```

Pointers to functions

- C doesn’t require that pointers point only to data;
- Function pointers point to memory addresses where functions are stored, e.g.

```
void (*fp) (void);
```

- A function’s name may be viewed as a constant pointer to a function.

Pointers to functions

```
#include <stdio.h>

void func_0() {
    printf("I am func_0():\n");
}

int main() {
    void (*fp) () = func_0;

    fp();
    (*fp)();
    (fp)();

    return 0;
}
```

```
I am func_0():
I am func_0():
I am func_0():
```

- Function name may be viewed as a const pointer.
- Parameters and return type must match
- Either form may be used to invoke the function

Arrays of Pointers to functions

```
#include <stdio.h>

void func_0(){ printf("I am func_0():\n"); }
void func_1(){ printf("I am func_1():\n"); }
void func_2(){ printf("I am func_2():\n"); }
void func_3(){ printf("I am func_3():\n"); }

int main() {
    void (*fp[5])() = {
        func_0,
        func_1,
        func_2,
        func_3
    };
    fp[4] = 0;

    int i=0;
    while (fp[i]) {
        fp[i]();
        i++;
    }
    return 0;
}
```

```
I am func_0():
I am func_1():
I am func_2():
I am func_3():
```

- We can also have arrays of function pointers
- In this particular example, we are using a “0” as a sentinel.

Pointers to functions

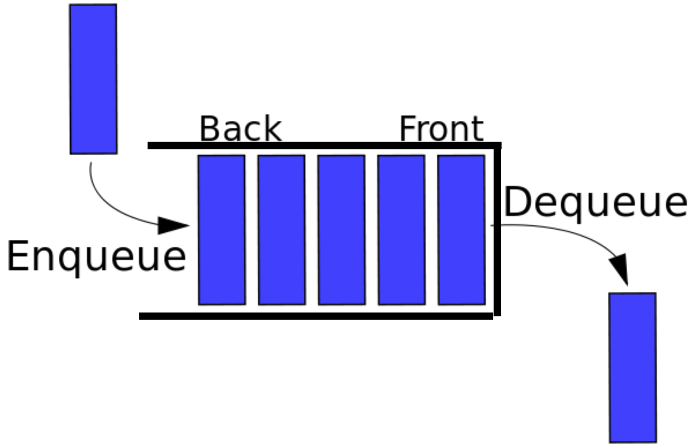
```
#include <stdio.h>

void func_0() {
    printf("I am func_0():\n");
}
struct Cobj{
    char *name;
    void (*printname)();
};
int main() {
    // create a c-style object
    struct Cobj myObj;

    // initialize the c-string and the methods
    myObj.name = "Sam Smith";
    myObj.printname = func_0;

    // Invoke the method
    myObj.printname();

    return 0;
}
```

```
I am func_0():
```

- Function pointers may be also declared inside structs
- They may be invoked like any method you would invoke in C++ or in Java
- **This how Linux implements object-oriented concepts, despite using C, a non object-oriented language!**

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

- Queue: a FIFO (first in, first out) data structure.
- Examples:
  - print jobs sent to a printer
  - Input from a keyboard is buffered into a stream using a fixed size FIFO.
  - TCP/IP packets waiting to be transmitted
- Implementation:
  - static: fixed size, implemented as array
  - dynamic: variable size, implemented as linked list



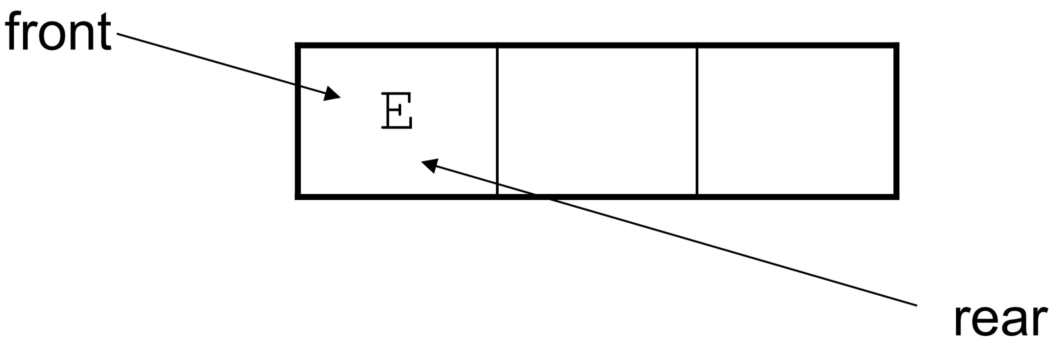
The Queue data structure - operations

- 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.

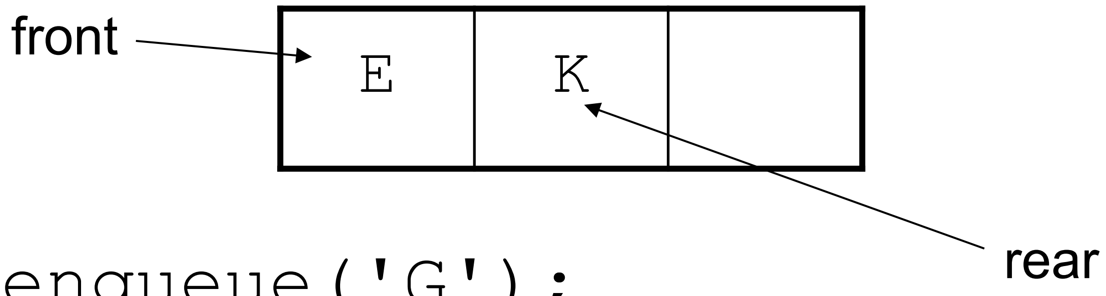
- A currently empty queue that can hold `char` values:

- `enqueue('E');`

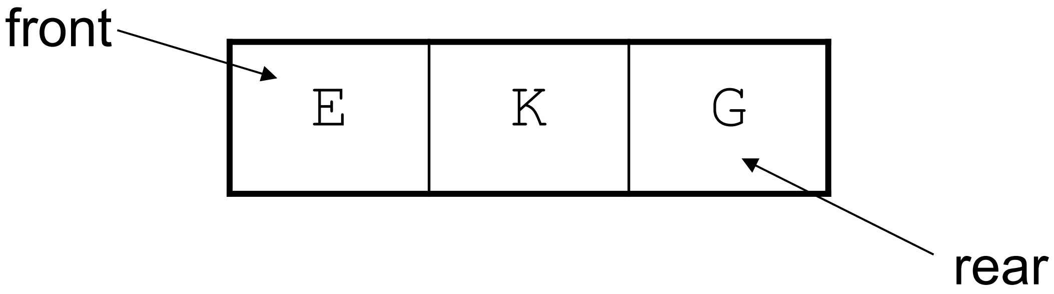


Queue operations – cont.

- `enqueue('K');`

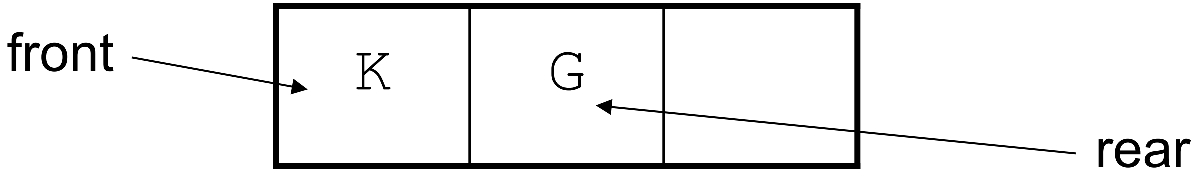


- `enqueue('G');`

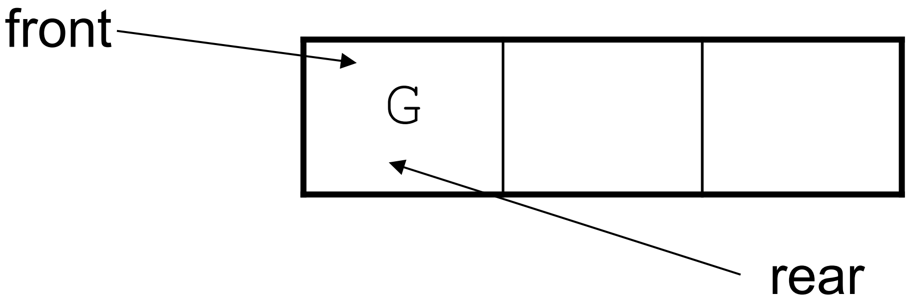


Queue operations – cont.

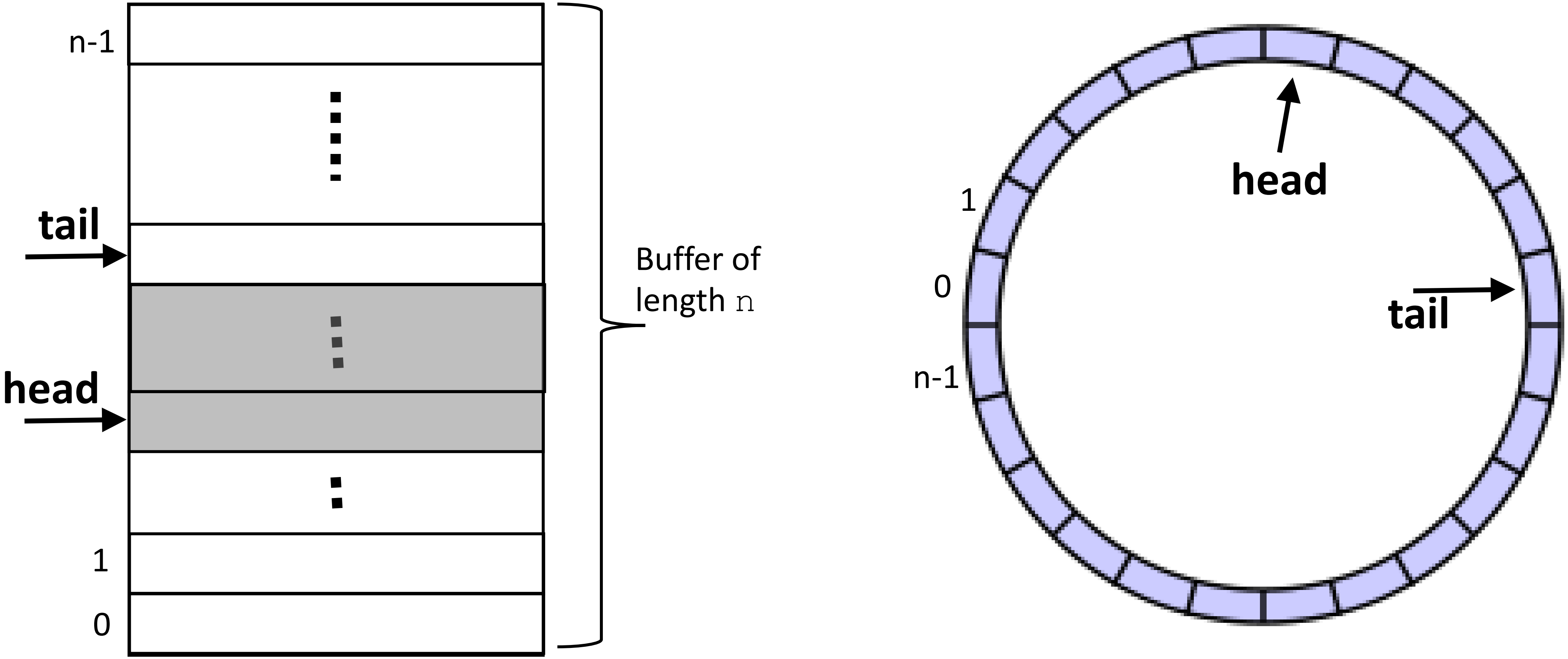
- `dequeue(); // remove E`



- `dequeue(); // remove K`



The Queue data structure – Fixed Buffer Implementations



The queue data structure - Design/algorithms

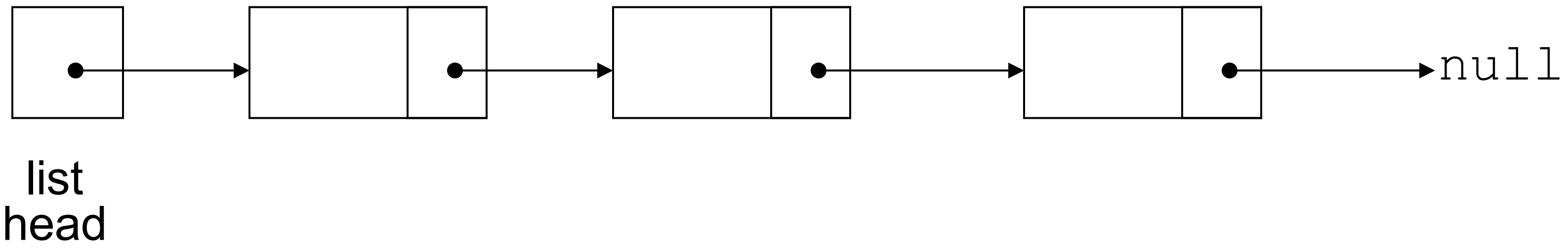
- Create a buffer of length n
- Create some variables : (all initialized to zero)
  - Use a variable (tail or write_index) to hold the index where new data should be written.
  - Use a variable (head or read_index) to point to the array index from which data may be read.
  - A count variable to hold 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

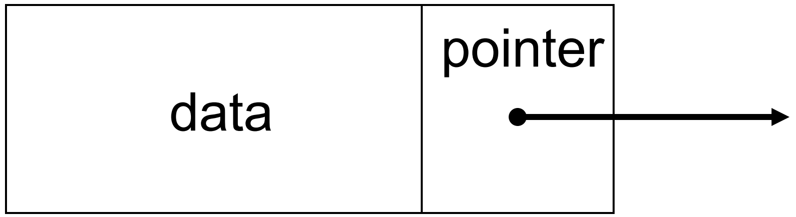
- Linked list: set of data structures (nodes) that contain references to other data structures
  - References may be addresses or array indices. In our OS class, the kernel uses addresses to point to the next node in the list.
- Nodes may be added to or removed from the linked list during execution



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 may be the data in the **process control block** (PCB), whereas the pointer is the next PCB (for the next process)



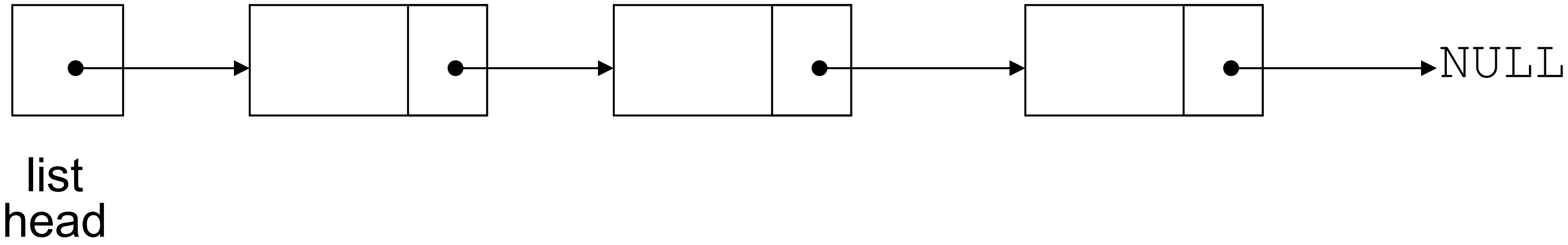
Declaring a Node

- Declare a node:

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

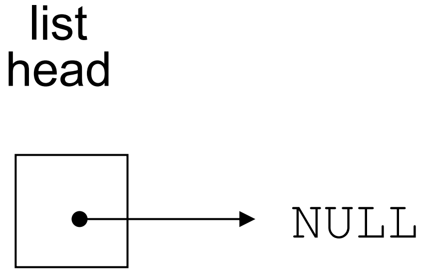
- Declaraing a `struct` does not allocate memory
  - Instantiating a node allocates memory, e.g.
    - `ListNode* ptr = new ListNode;`

Linked List Organization



Linked lists may contain 0 or more nodes:

- Has a list head to point to first node
- Last node points to `NULL`



<u>An empty list</u> contains 0 nodes,

- The list head points to `NULL`

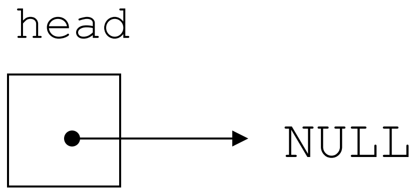
Creating a Linked List (an empty one)

- 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.



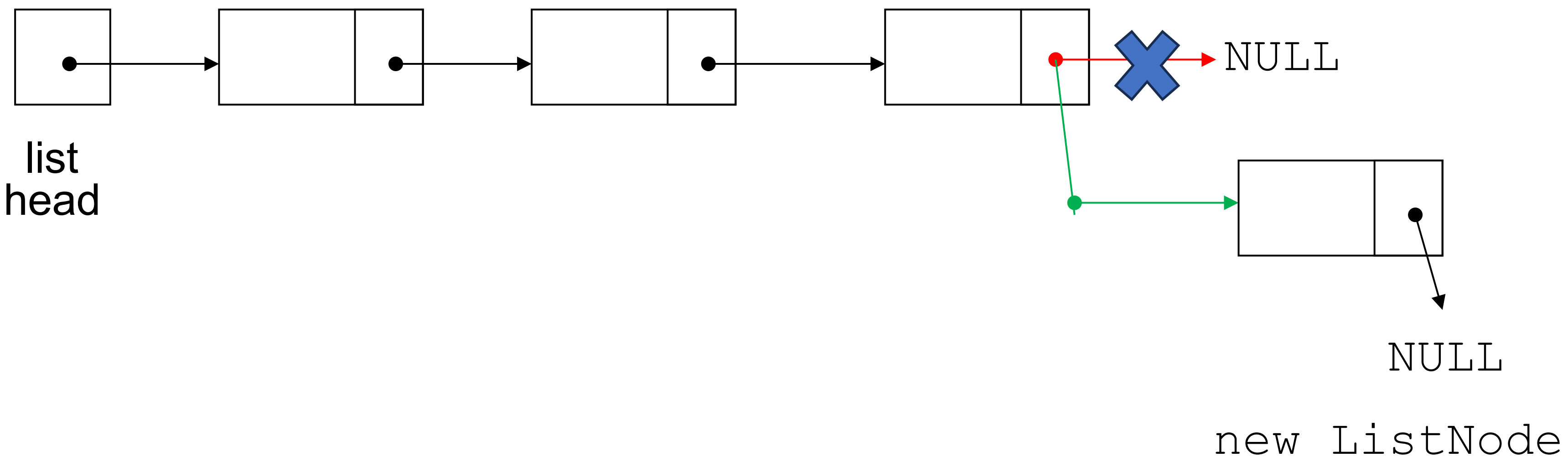
The Null Pointer

- 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:

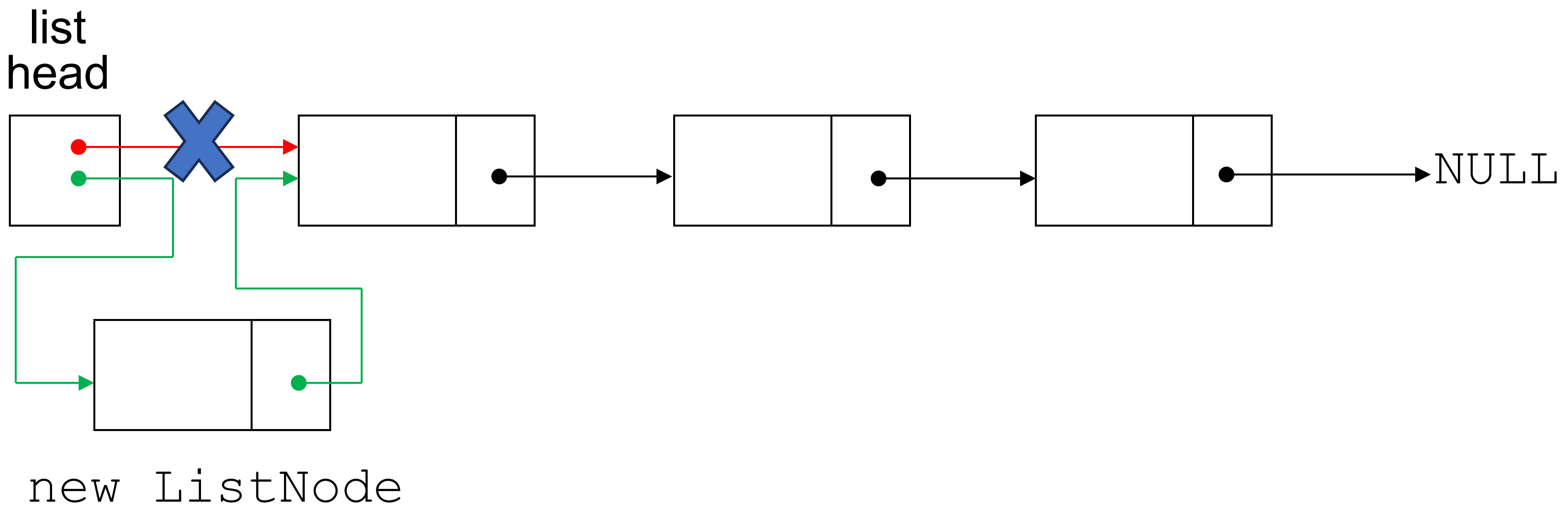
Linked List Operations

- 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

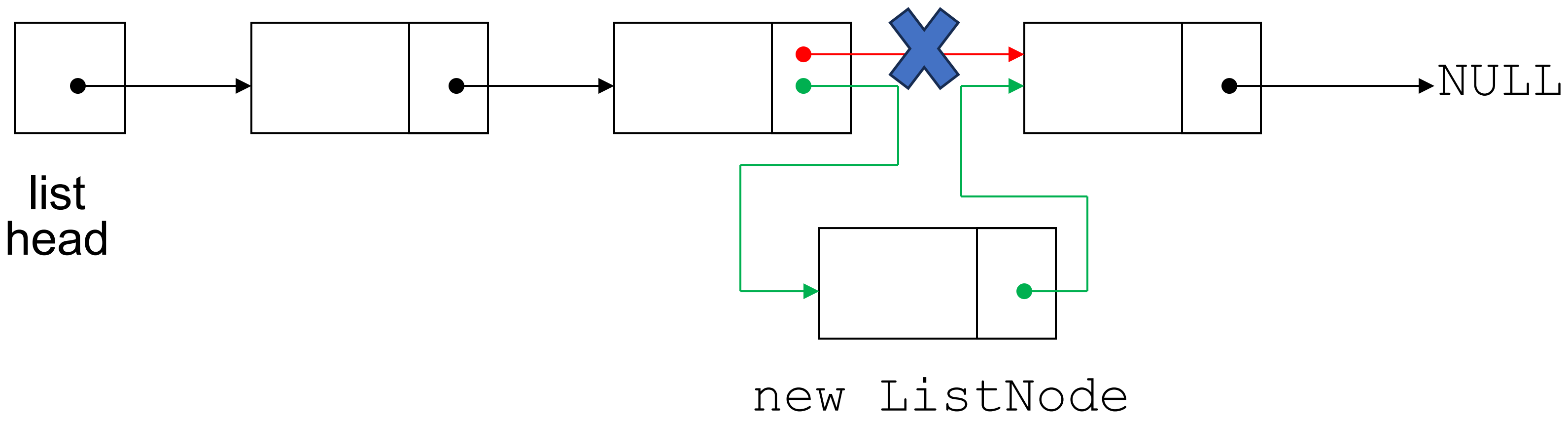
Linked List operations – appending to tail



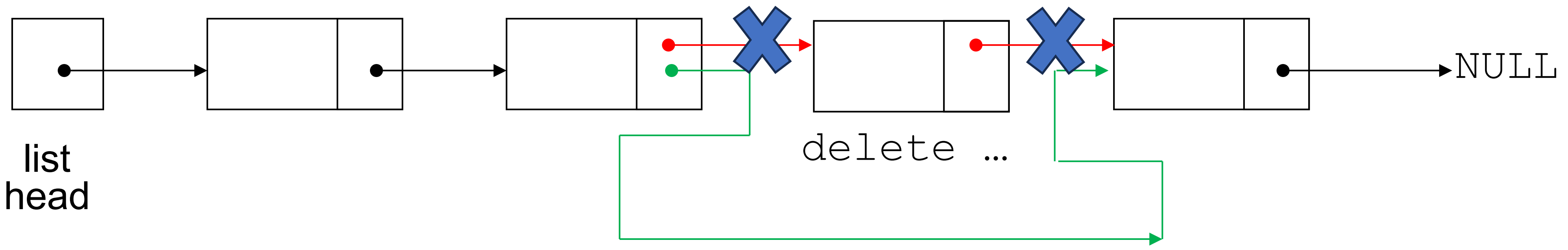
Linked List Operations – appending to head



Linked List Operations – inserting a node

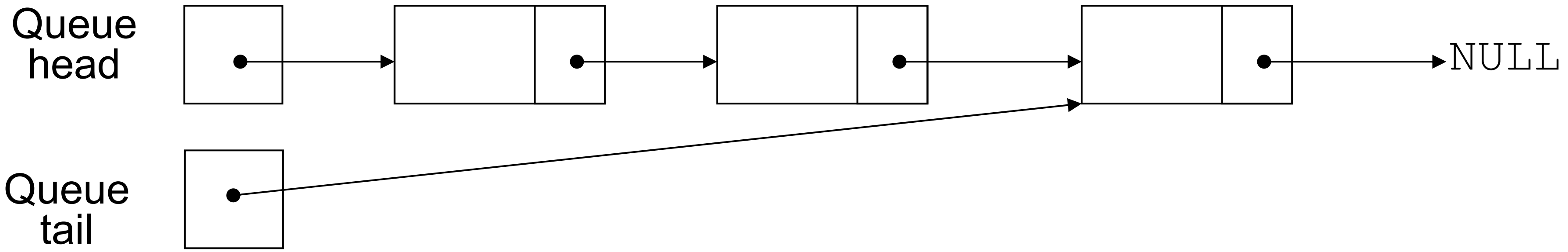


Linked List Operations – removing a node



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



Chapter 3: Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

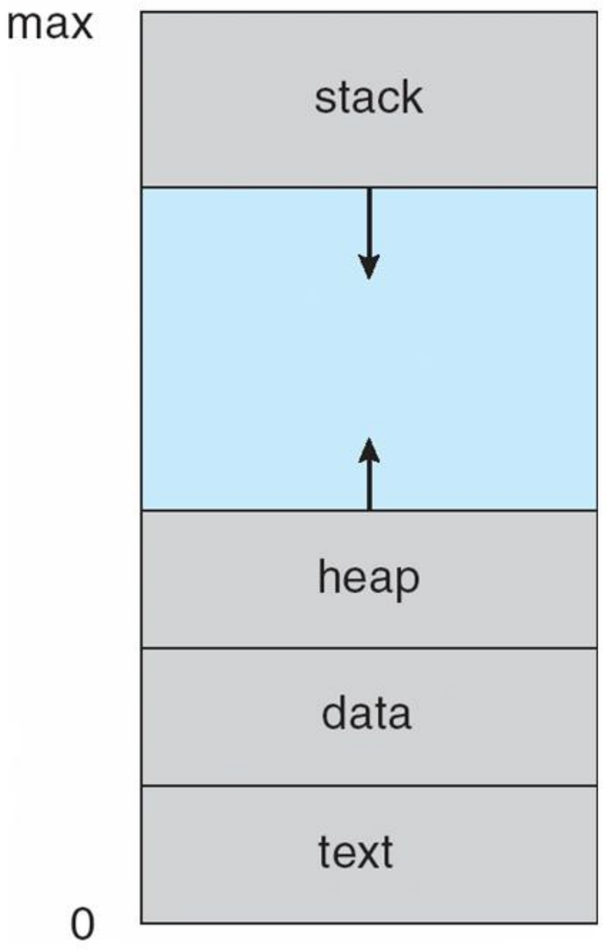
3.1 The Process Concept

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs, processes** or **tasks**
- We use the terms ***job, task*** and ***process*** almost interchangeably
  - In conventional full-fledged operating systems, i.e. those running workstations or servers (not embedded systems running FreeRTOS):

    process = job = task
- **Process** – a program in execution; process execution generally (but not always) progresses in a sequential fashion.

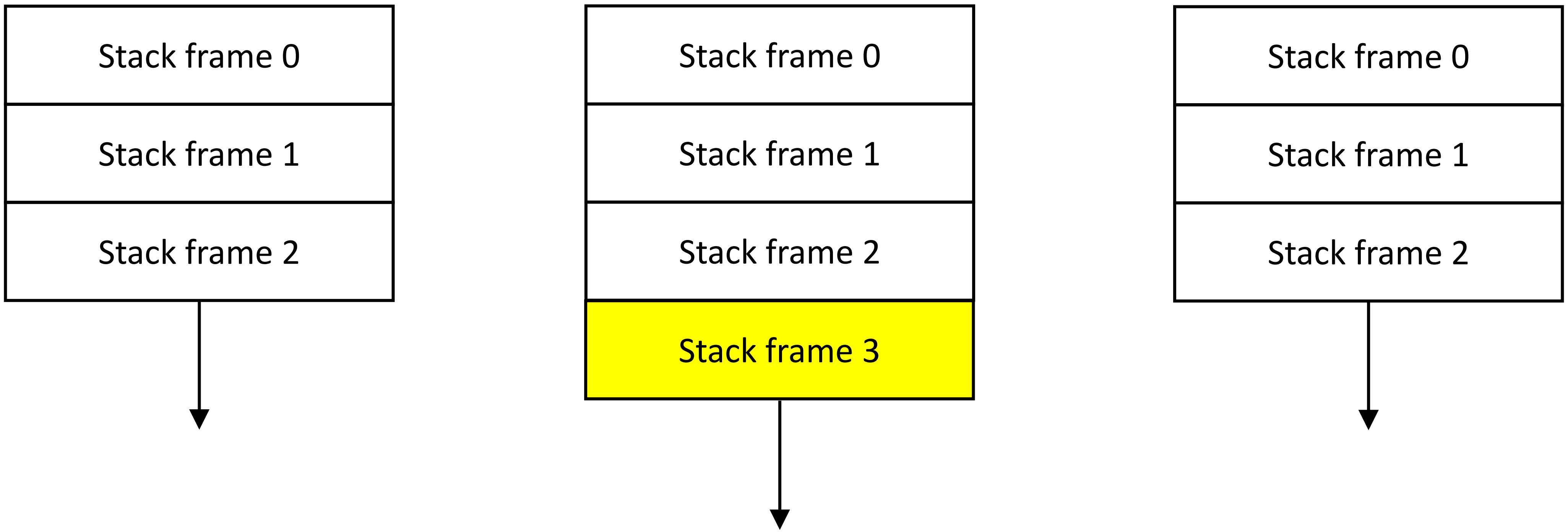
The Process Concept – cont.

- Area occupied in main memory is divided into multiple sections:
  - The program code, also called **text section**
  - **Data section** containing global variables
    - Initialized sections (aka .data section), followed by
    - Uninitialized sections (aka .bss section)
  - **Stack:** The **call stack** contains temporary data:
    - Saved CPU registers (including return addresses)
    - Function parameters
    - Local variables
  - **Heap** containing memory dynamically allocated during run time. Grows opposite to the stack
    - e.g. using new/delete (in C++ or Java)
    - Malloc/free in C
- A process also occupies/uses CPU registers (not shown on Fig.).



The Call Stack - example

```
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int myfunc(int x) {
    int y;
    y = x * x + 2 * x + 5;
    return y;
}
int main() {
    int z;
    z = myfunc(2);
    cout << z << endl;
}
```



Before calling myfunc()

While myfunc() is executing:

- CPU registers are saved into frame 3
- Parameter x is pushed into stack frame 3
- Local variable y is allocated in frame 3

After myfunc() executed:

- CPU registers are restored from frame 3
- Parameter x is deallocated
- Local variable y deallocated
- Stack frame is no more

The Process Concept (Cont.)

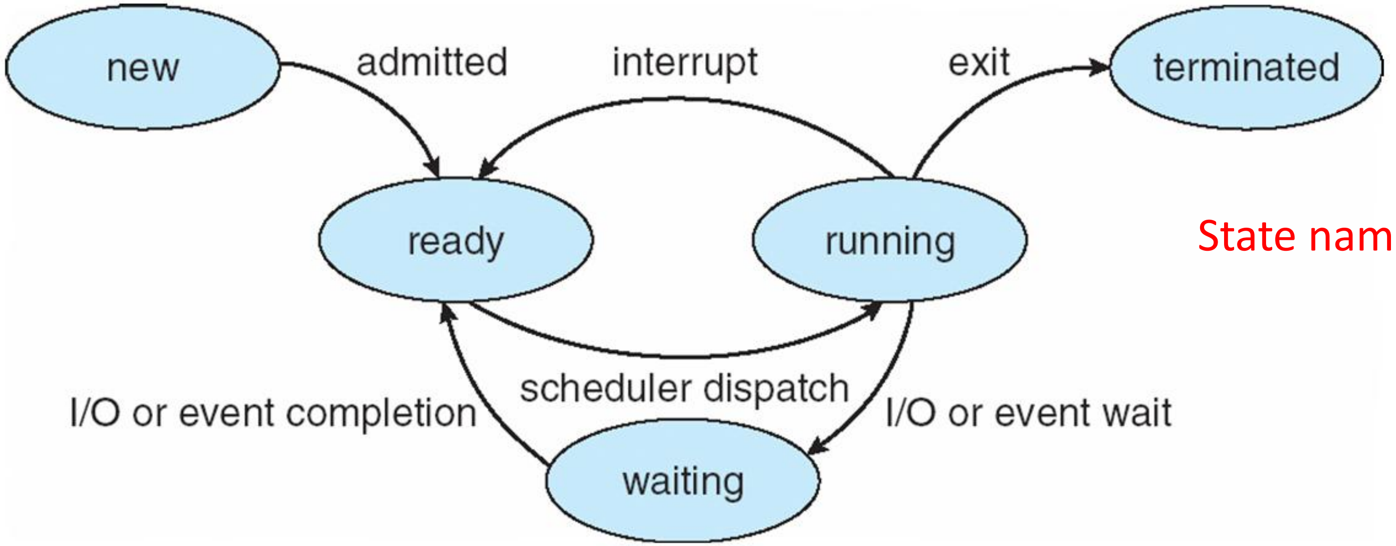
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active.*** A program becomes a process when:
  - Its executable file is loaded into main memory and
  - is registered with the scheduler for execution.
- Execution of program is usually started via:
  - GUI mouse clicks
  - Command line entry of its name, etc.
- One program can be instantiated as several processes
  - Consider multiple users executing the same program

Process State

- As a process executes, it changes **state**
  - **new**: The process is being created

  - **ready**: The process is waiting to be assigned to a processor
  - **running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur

  - **terminated**: The process has finished execution



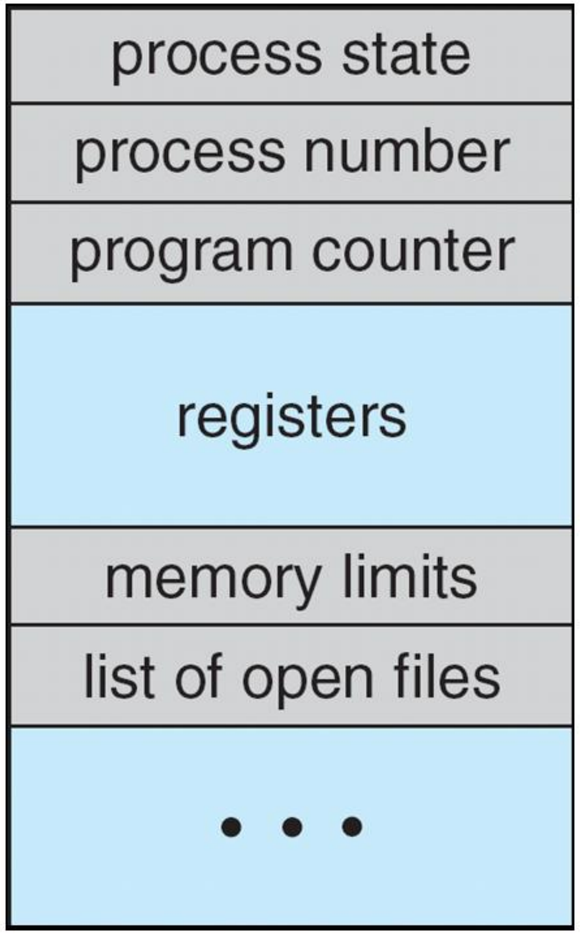
State names are generic

Process Control Block (PCB)

Information associated with each process

(also called **task control block**)

- Process ID
- Process state – running, waiting, etc
- CPU registers – contents of all process-centric registers **including the program counter** (which contains location of next instruction to execute)
- CPU scheduling information - priorities, scheduling queue pointers, etc.
- Memory-management information – memory allocated to the process (base and limit registers, page/segment tables, etc.)
- Accounting information – CPU and real time used, time limits
- Process numbers of parents or children
- Allocated resources – I/O devices allocated to process, list of open files



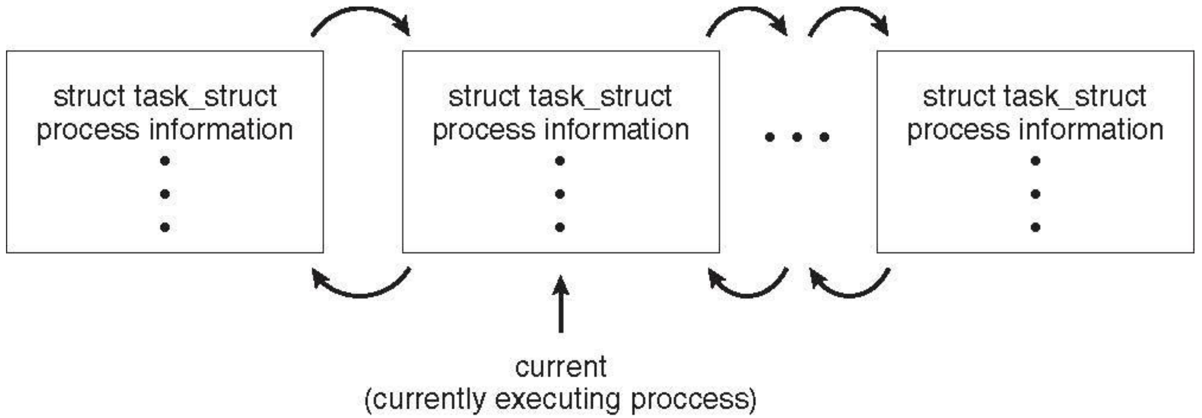
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple flows of executions == **threads**
    - Running concurrently OR
    - Running in parallel (in case of multiple processor cores)
- Must then have storage for thread details, and multiple program counters in PCB
- More about threads in the next chapter.

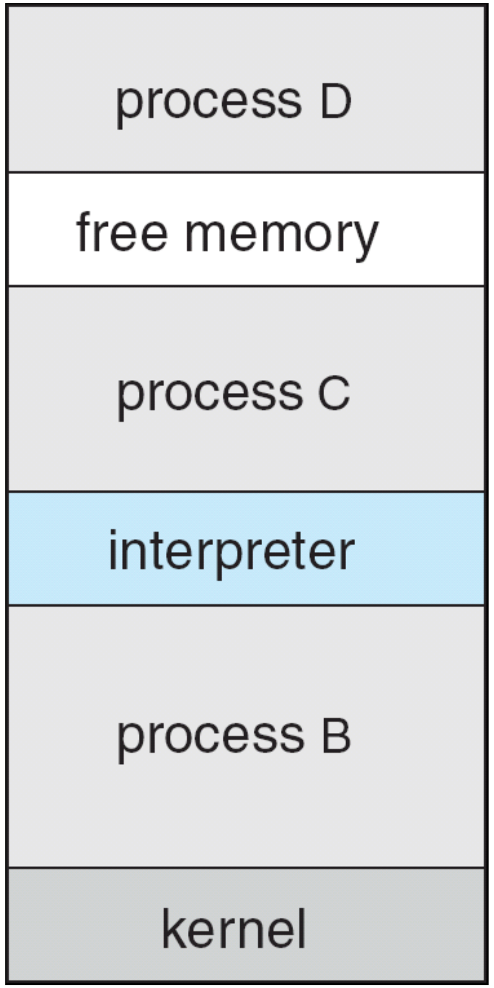
Process Representation in Linux

Represented by the C structure `task_struct`

```
pid_t pid; /* process identifier */
long state; /* state of the process */
long nice;

unsigned long policy;

struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct task_struct *next_task, *prev_task;

struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Where is the PCB for process C ?



main memory