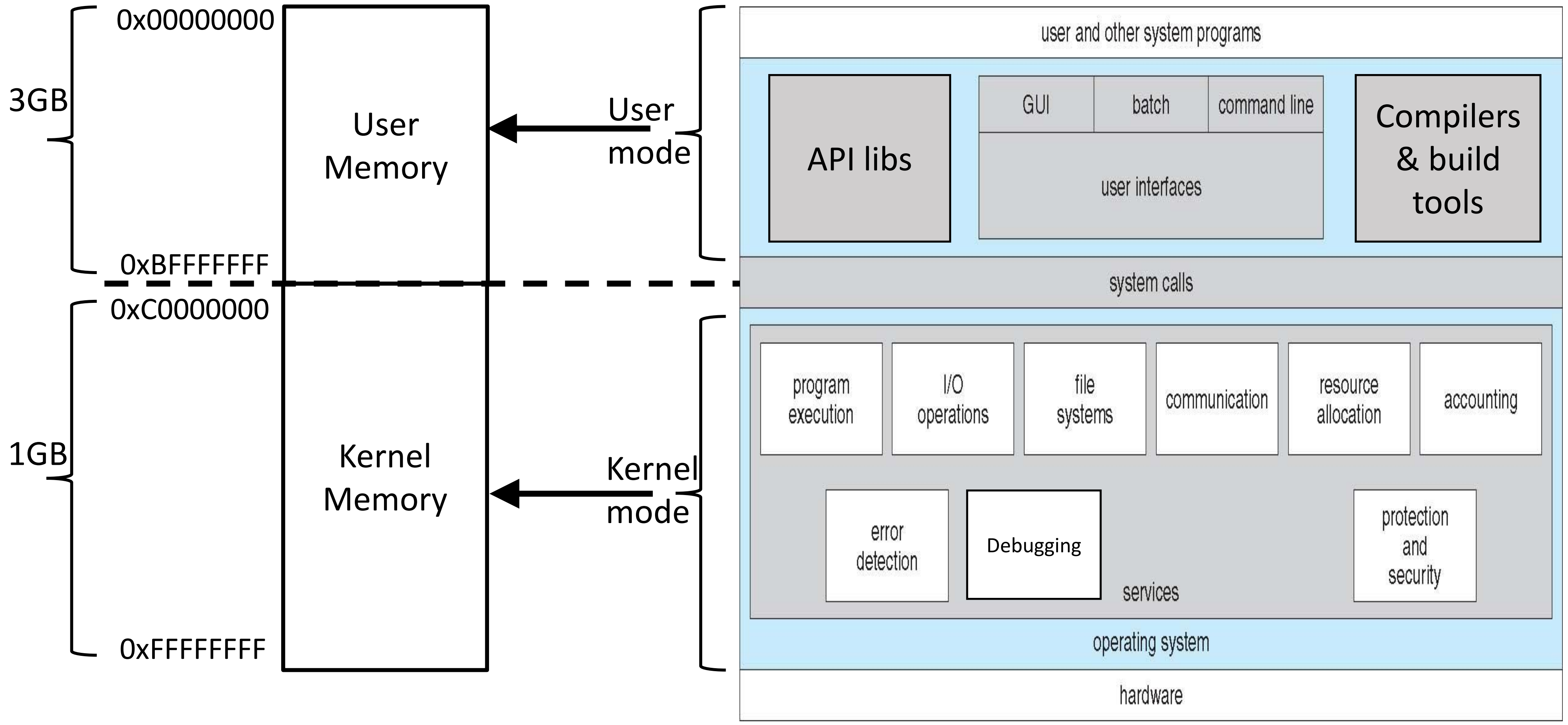
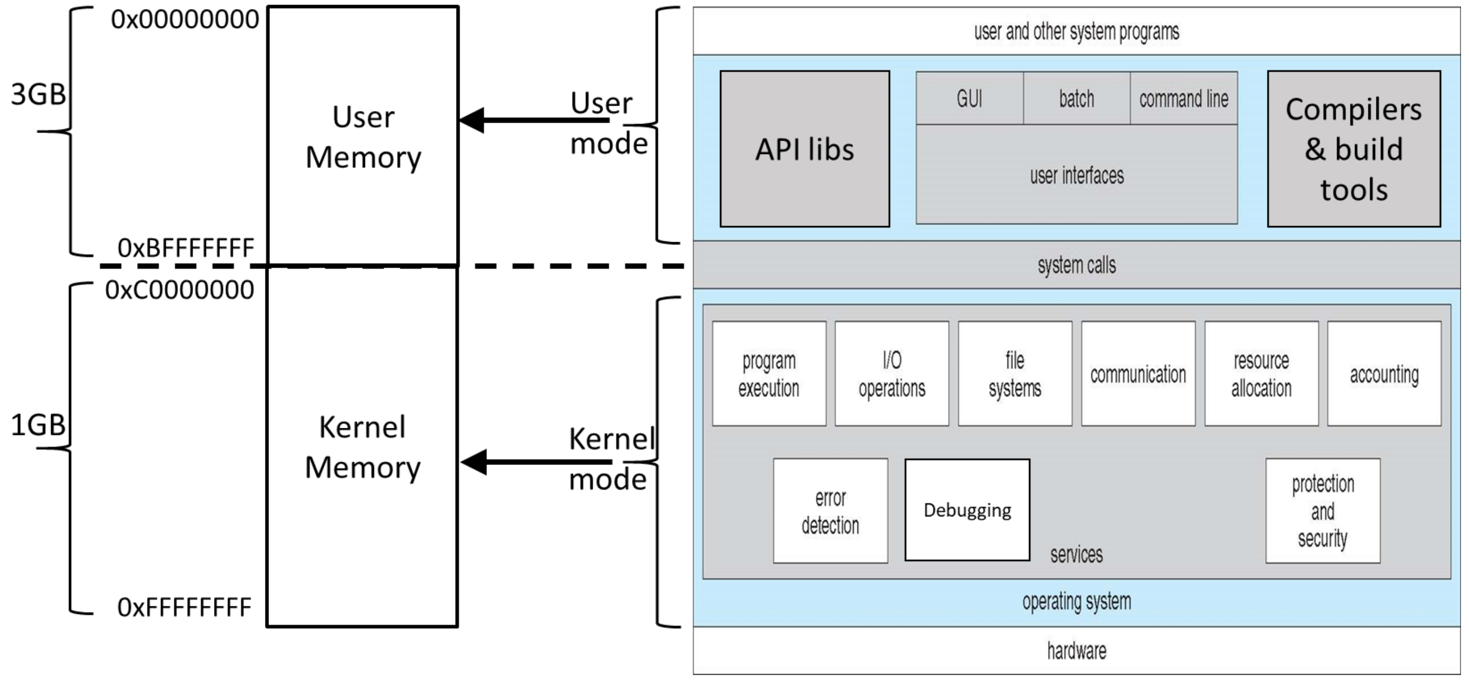
kernel space / user space – 32-bit linux example



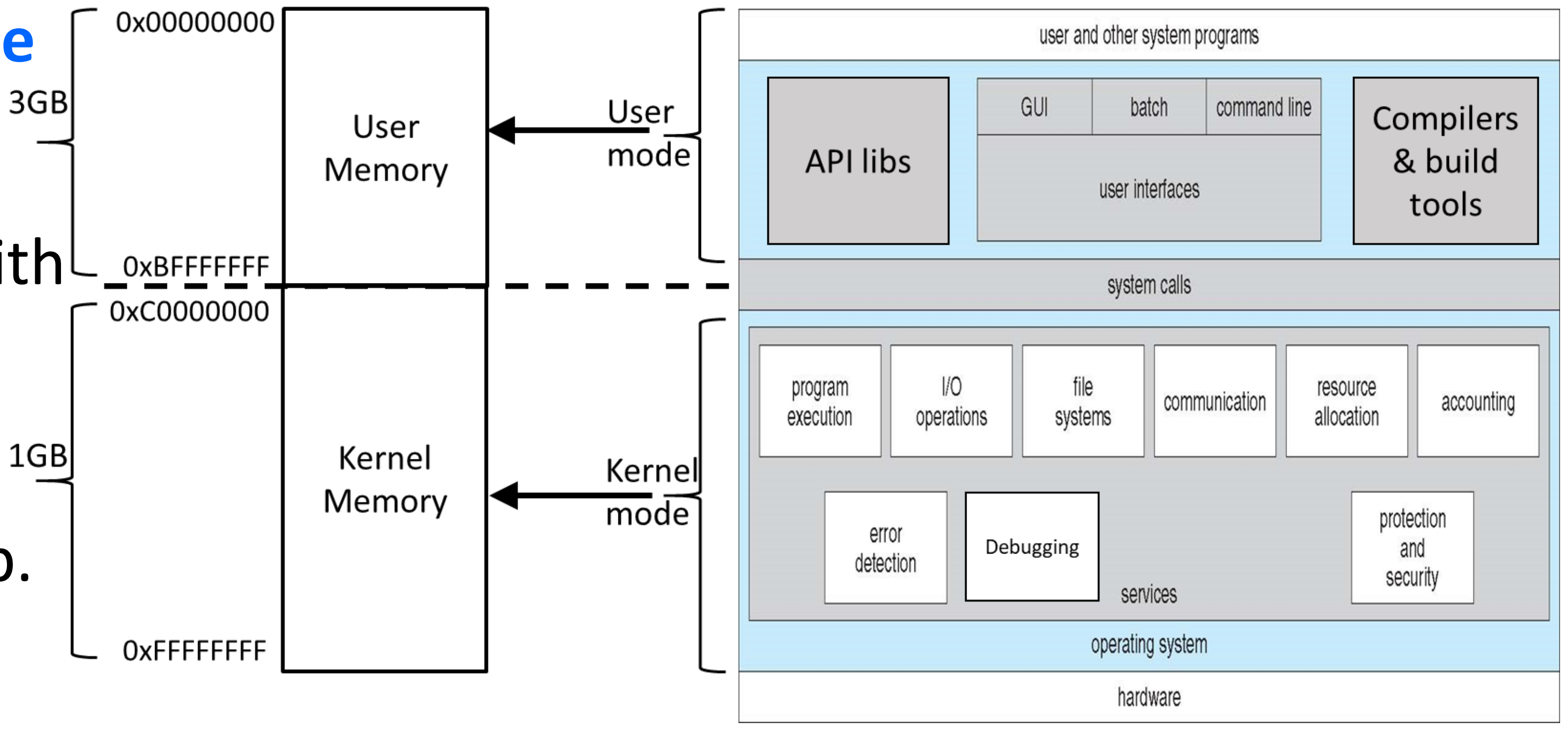
kernel space / user space – 32-bit linux example

- 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).

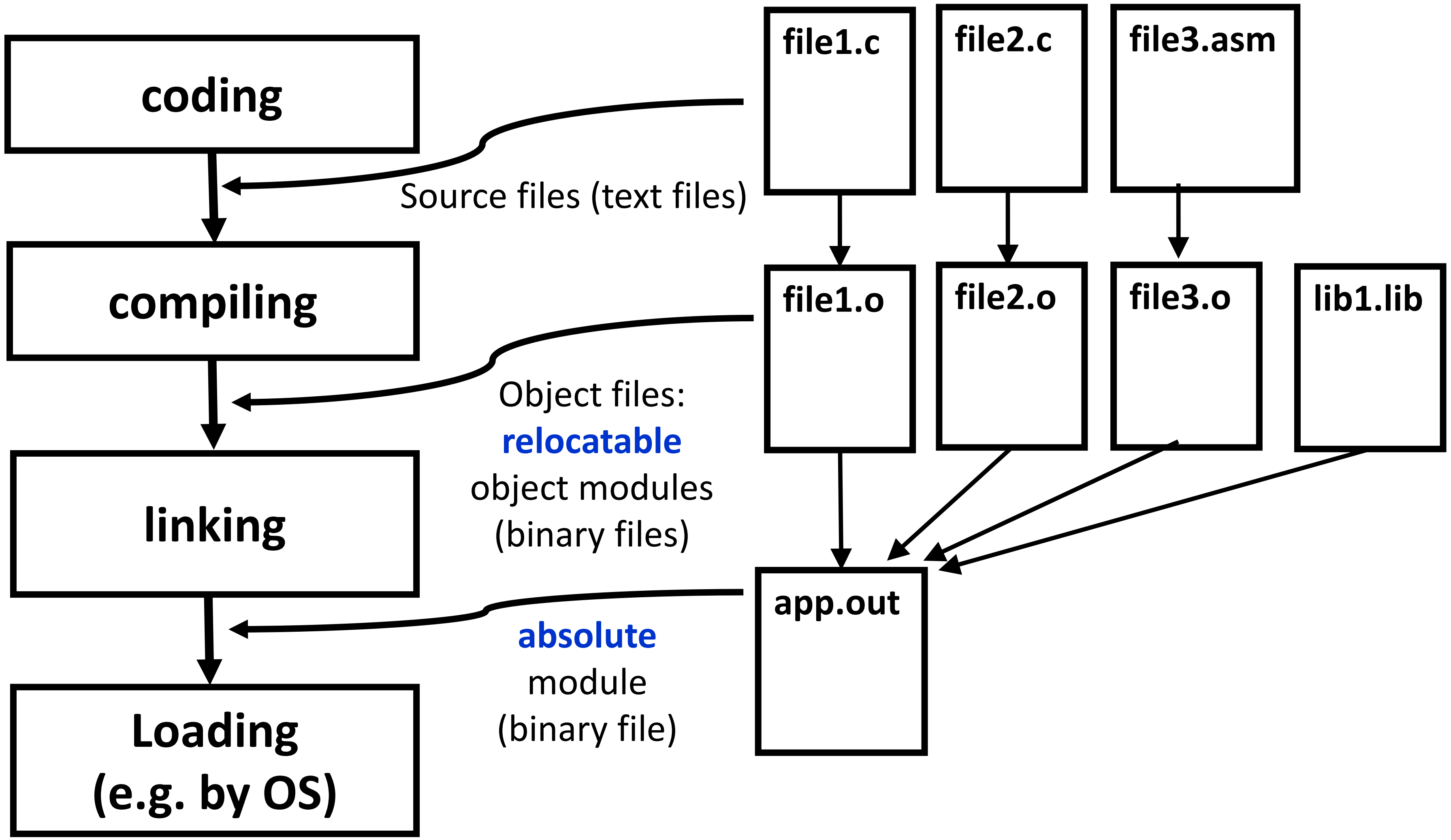


kernel space / user space – 32-bit linux example

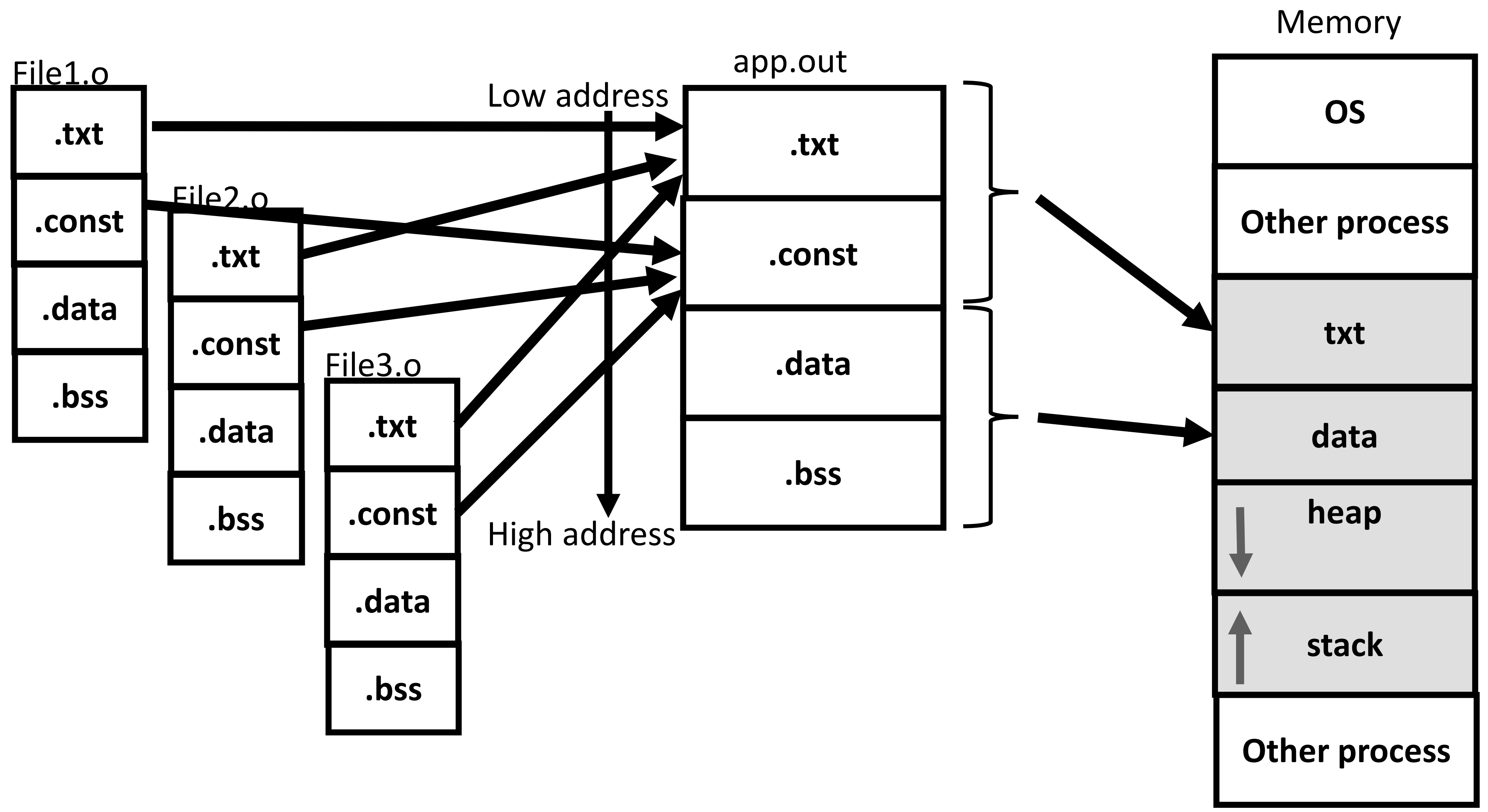
- Thus, the only way for a user program to invoke kernel functions **is via the system call interface**, in which a trap machine instruction is invoked, with the correct paramaters passed via registers/memory/stack prior to invoking the trap.

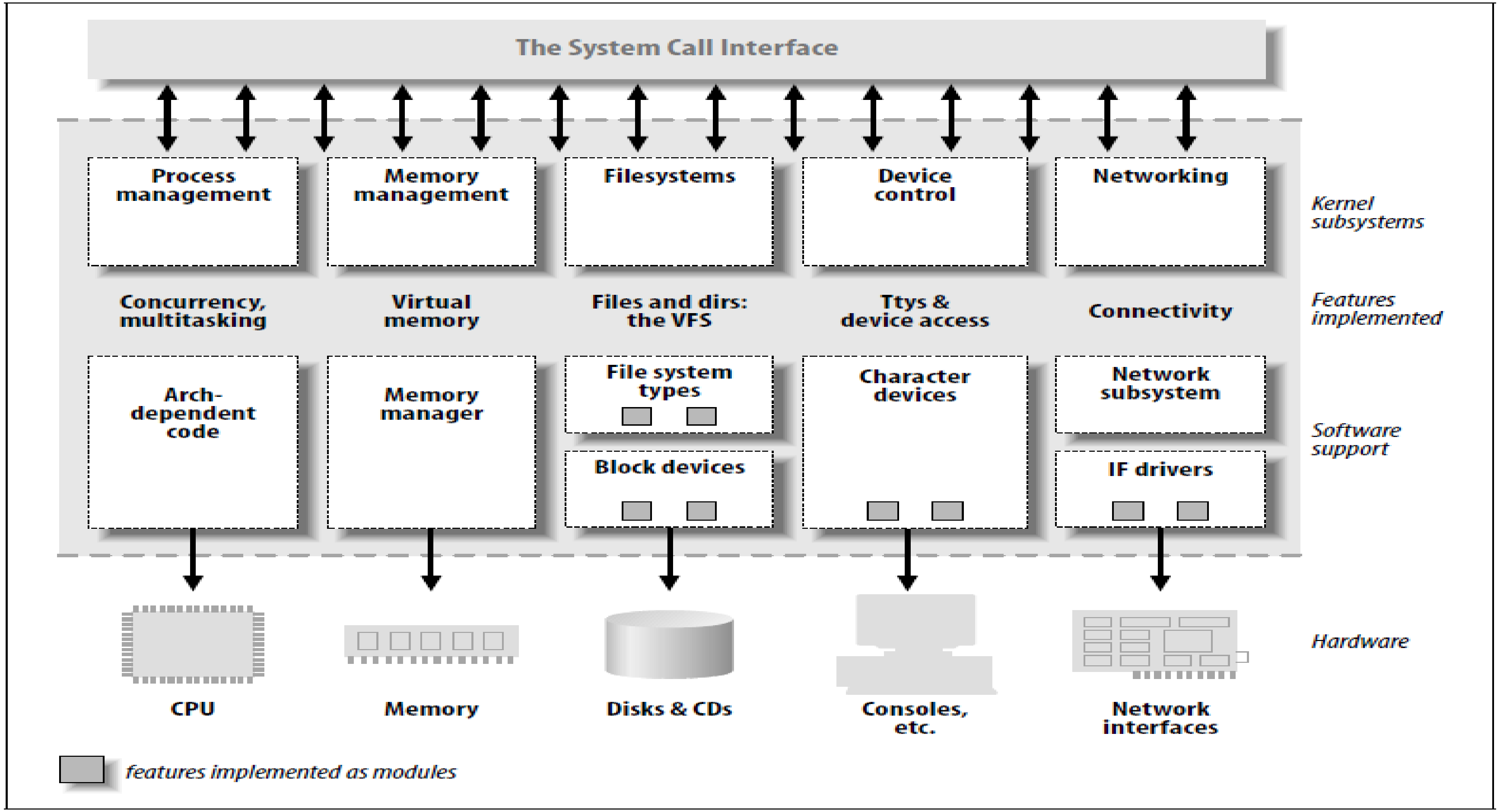


Program translation – (user-mode programs)

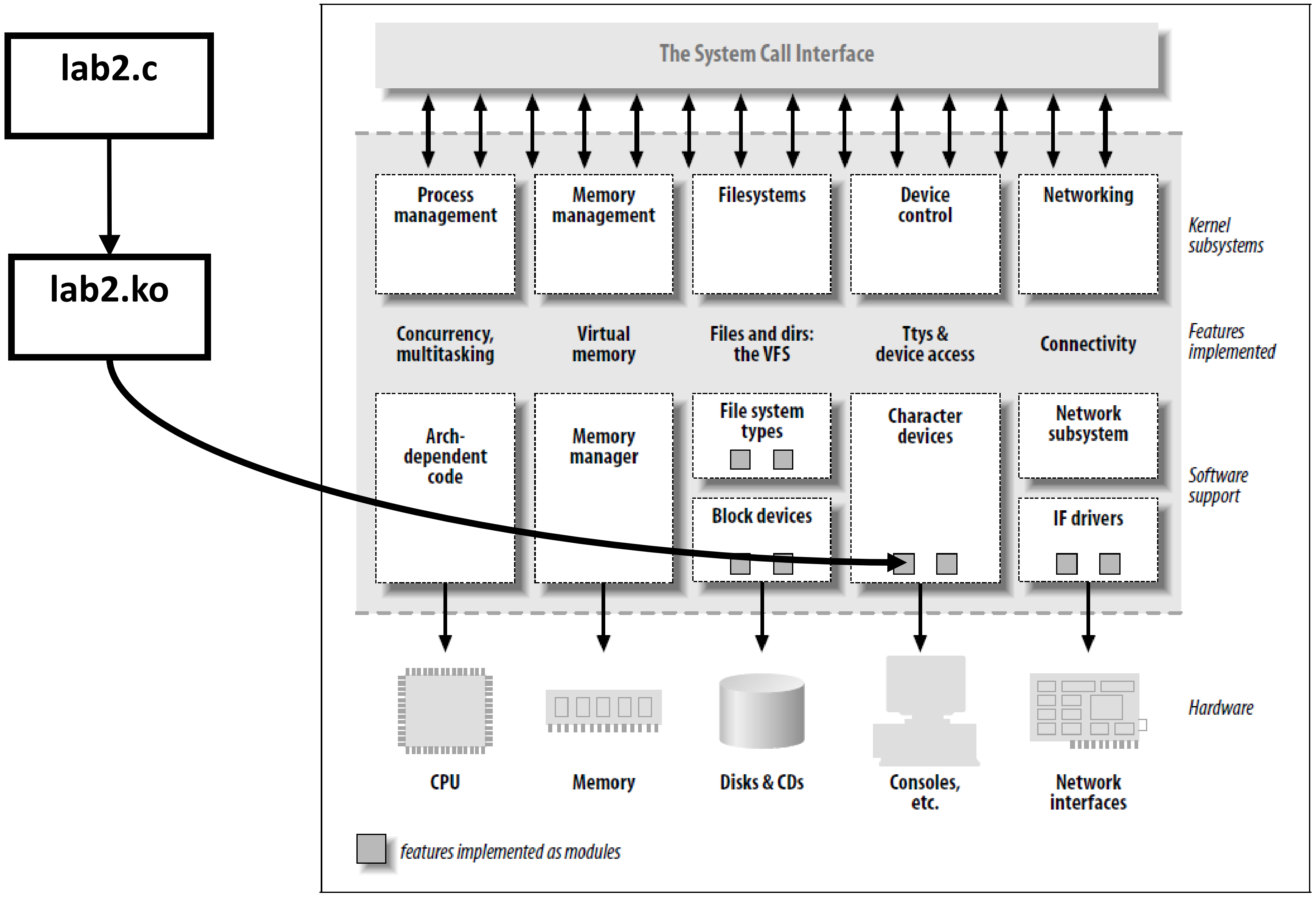


Program translation – linking + loading (user-mode programs)




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/

2.4 Types of System Calls

- Process control (and support)
  - create process, terminate process, end or abort
  - load, execute
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
  - Dump memory if error
  - **Debugger** for determining bugs in a program
  - **Locks** for managing access to shared data between processes

Types of System Calls (cont.)

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes (e.g. filename, type, protection code, accounting info, etc.)
- Device management
  - request device, release device (to exclusive access to a device)
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices (make it look like a file on the file system)

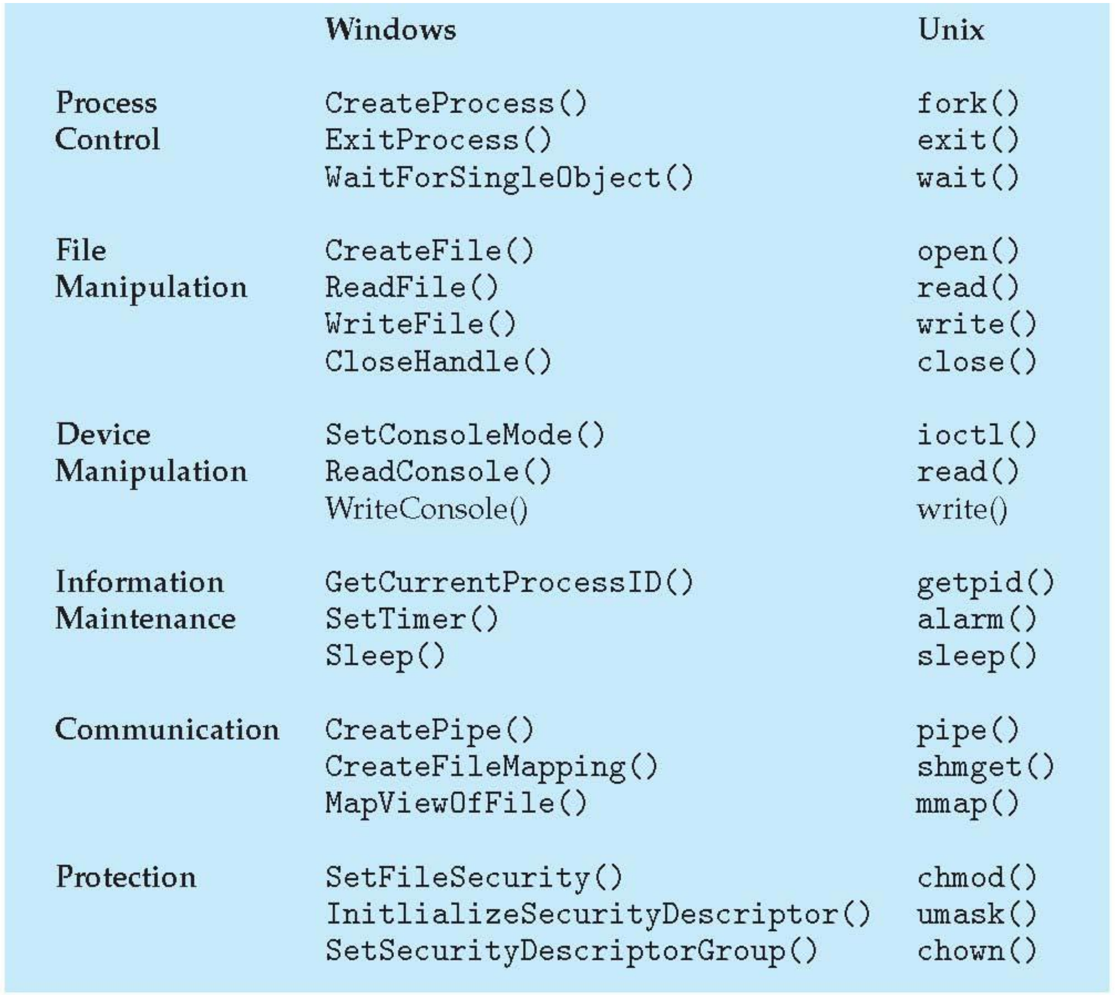
Types of System Calls (Cont.)

- Information
  - get system time or date, set time or date
  - get system data (e.g. OS version, number of users, memory or disk space, etc.).
  - Get the time profile of a process (via regular timer interrupts)
  - get and set process, file, or device attributes
- Communications
  - create, delete, open and close communication connection
  - In **message passing model,** processes can send and receive messages to **host name** or **process name** (name is first translated to a process ID (PID))
    - A process waiting for a connection request is a **server or a Daemon**
  - In **Shared-memory model,** processes create and gain access to memory regions shared with other processes.
  - Message passing is useful for small messages, while shared memory is preferred in larger messages.

Types of System Calls (Cont.)

- Protection (of **resources**)
  - Control access to resources (e.g. files, disks, memory locations, etc.)
  - Get and set permissions of resources
  - Allow and deny user access of resources.

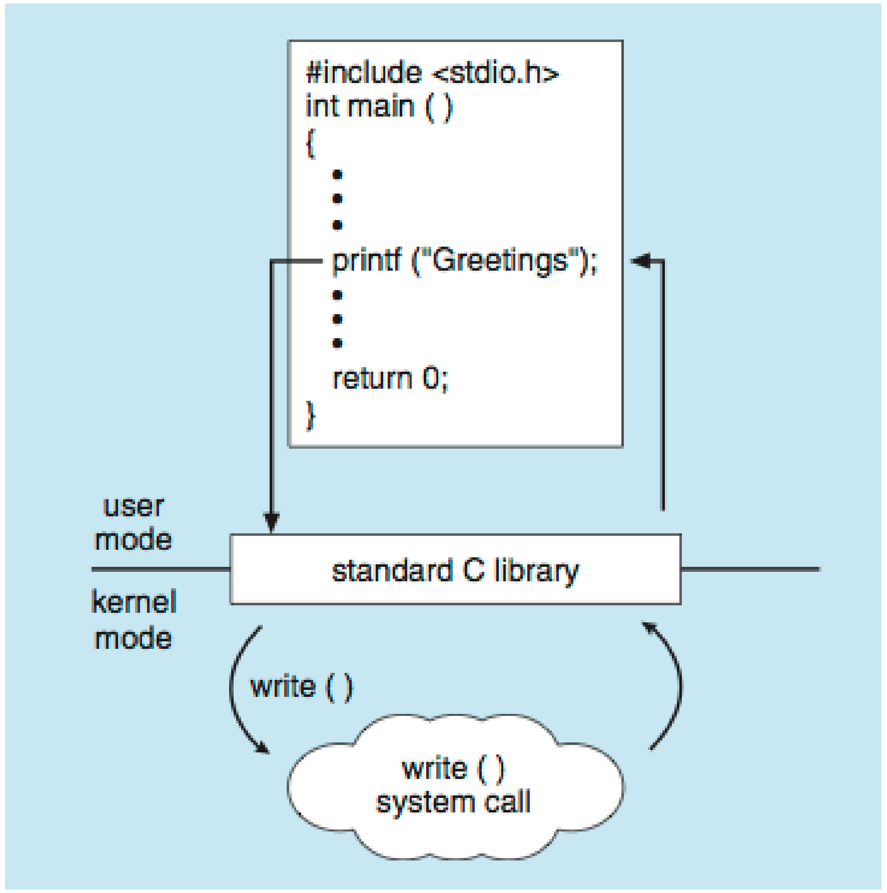
Examples of Windows and Unix System Calls



	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Standard C Library Example

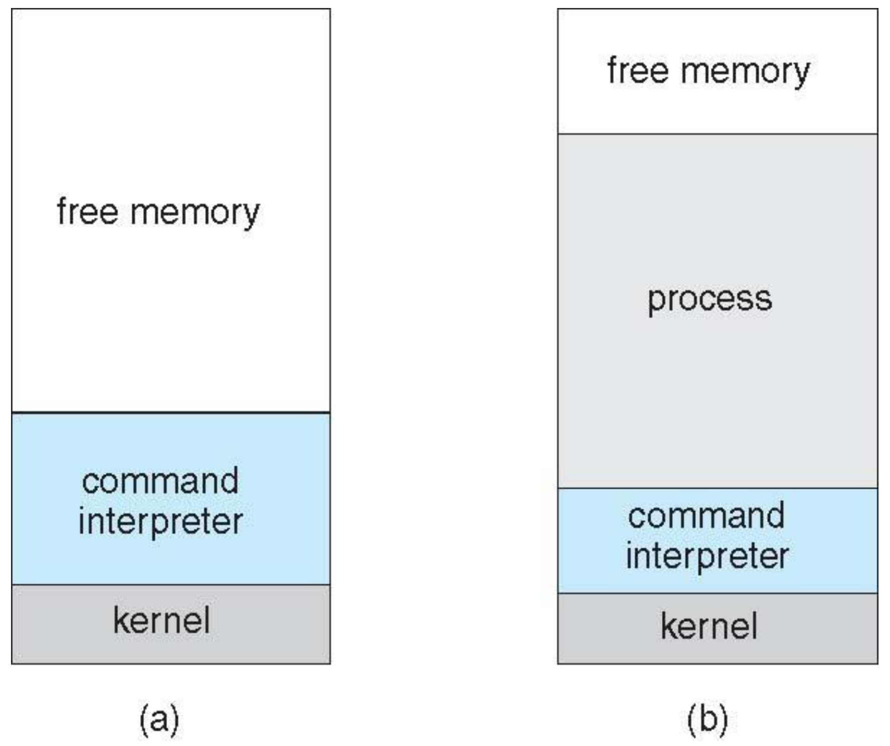
- C program invoking printf() library call, which calls write() system call



```
#include <stdio.h>
int main ( )
{
   •
   •
   •
   printf ("Greetings");
   •
   •
   •
   return 0;
}
```

Process control example - MS-DOS

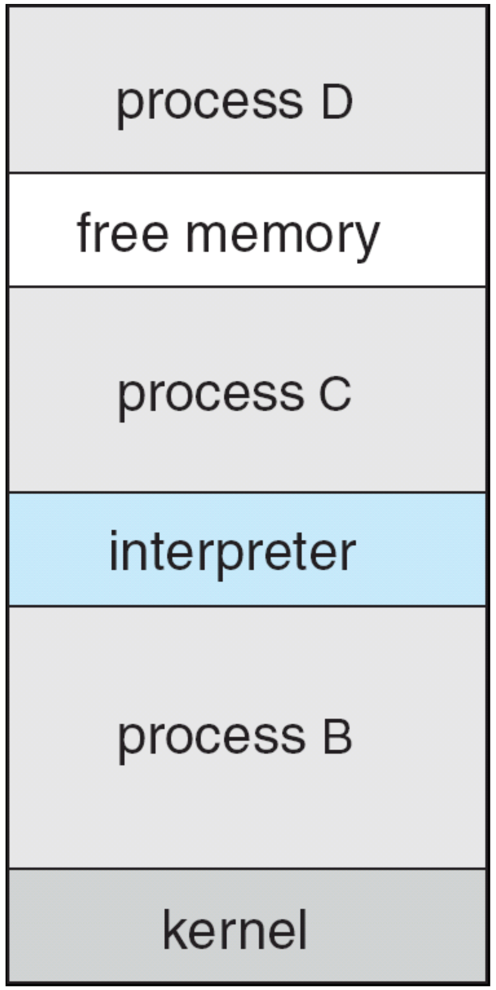
- Single-tasking
- Shell invoked when system booted
- Simple method to run program
- Initial versions had a single memory space
- Loads program into memory, overwriting all but the kernel. Only a minimal set of shell routines are kept.
- Program exit -> remainder of shell reloaded



At system startup   running a program

Process control example - FreeBSD

- Unix variant
- Multitasking
- User login -> invoke user's choice of shell
- Shell executes fork() system call to create process
  - Calls exec() to load program into process
  - Shell waits for process to terminate or continues with user commands
- Process exits with:
  - code = 0 – no error
  - code > 0 – error code



2.5 System Programs and utilites

- Most users' view of the operation system is defined by system programs, not the actual system calls
  - Provide a convenient environment for program execution
  - Some of them are simply user interfaces to system calls; others are considerably more complex.
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Provide **simple info** such as date, time, amount of available memory, disk space or number of users
  - May provide **detailed info** such as performance, logging, and debugging information
  - Some systems implement a **registry** - used to store and retrieve configuration information

System Programs (Cont.)

- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text (e.g. grep)
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided. May also provide a debugging system (for higher level as well as machine language).
- **Program loading and execution**- provides ability to load a program into main memory, as well as linking it to shared libraries (.dll in windows or .so in linux/unix)
- **Communications** - Provide the **mechanism** for creating virtual connections among processes, users, and computer systems.

System Programs (Cont.)

- **Background Services**
    - These are system programs that are run automatically. Known as **services**, **subsystems**, **daemons**
    - Some run at system startup (or boot) and terminate ast some point later.
    - Others run from system boot to shutdown
    - Ex: A network daemon may be listening to connection requests, and then connects them to the appropriate service.
    - Provide other facilities like disk checking, error logging, printing
    - Run in user context, not kernel context

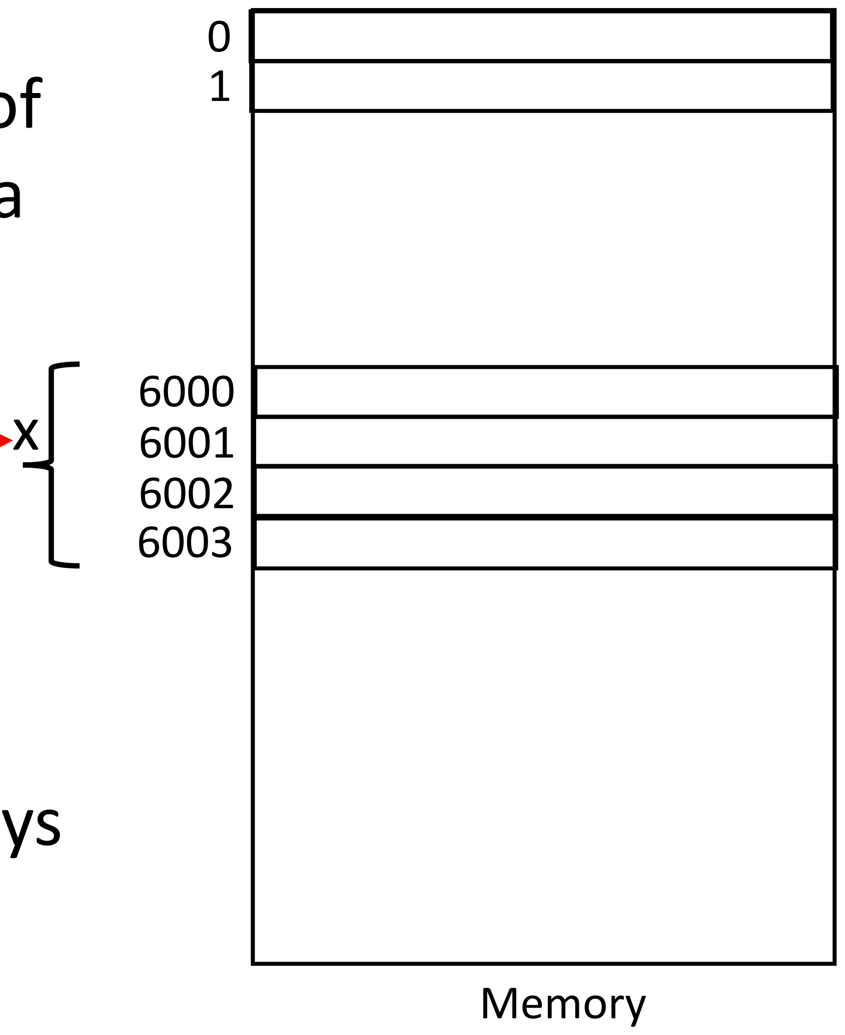
Application programs

- Don’t pertain to system
- Run by users
- Not typically considered part of OS
- Launched by command line, mouse click, finger poke
- e.g. web browsers, email clients, word processors, games, etc.

C-language review - Pointers

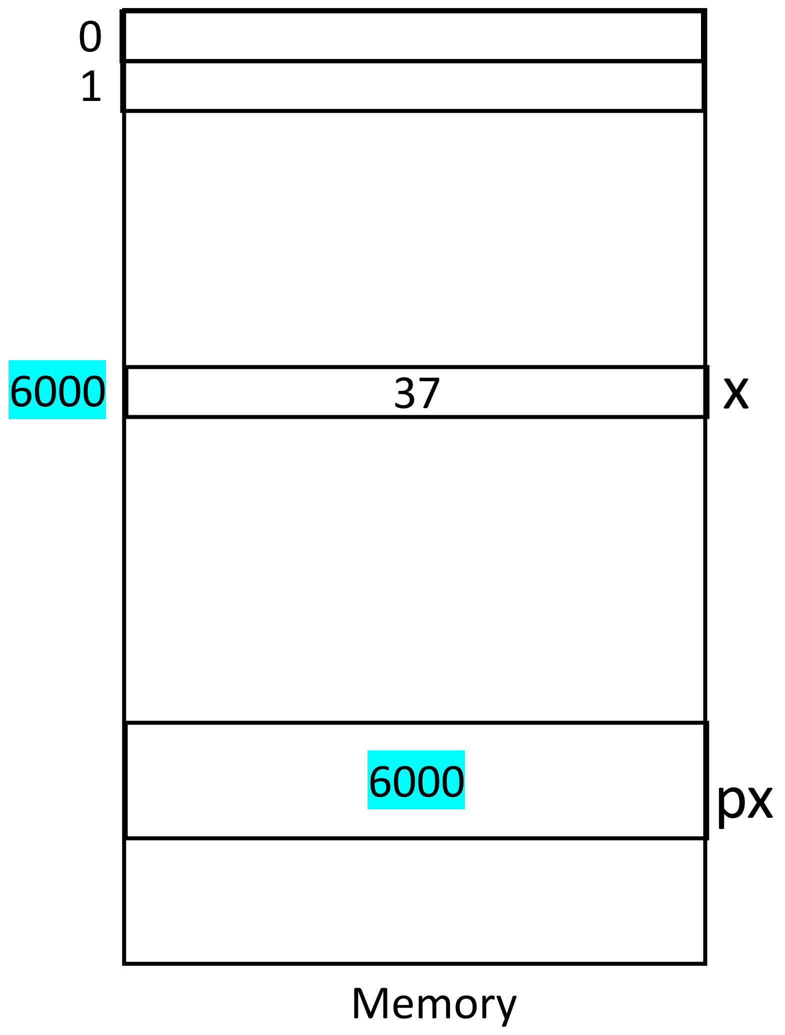
- Declaring a C variable allocates a number of memory cells (or bytes) and assigns them a name (the name of the variable).
- Memory is organized into bytes (or cells), where each has a unique address.
- When a variable is allocated a certain number of bytes in memory, they are always contiguous, for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

```
int x;
```



Pointers (cont.)

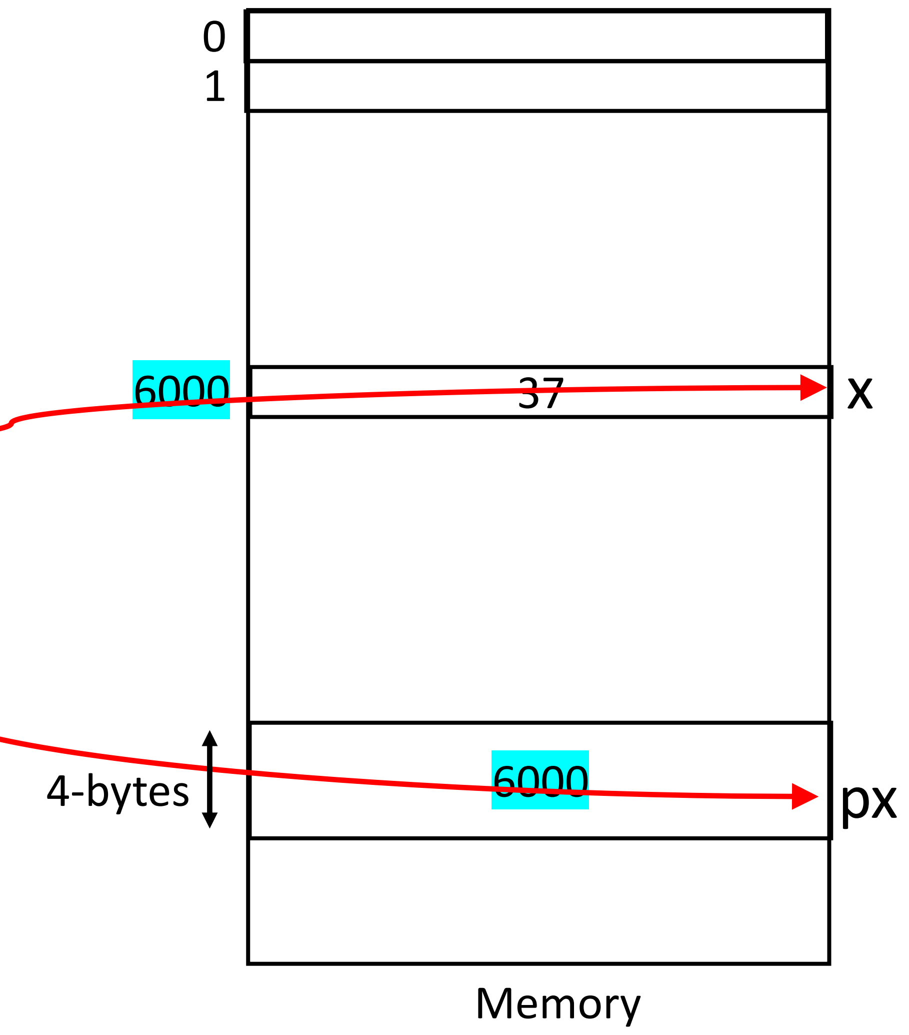
- In many cases, your program may need to know the memory address of your variable and may also need to access a variable using its address instead of its name.
  - Note: An address is always a byte-address (e.g. address 6000 tells you integer x is preceded by 6000 bytes before it, NOT 6000 integers,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

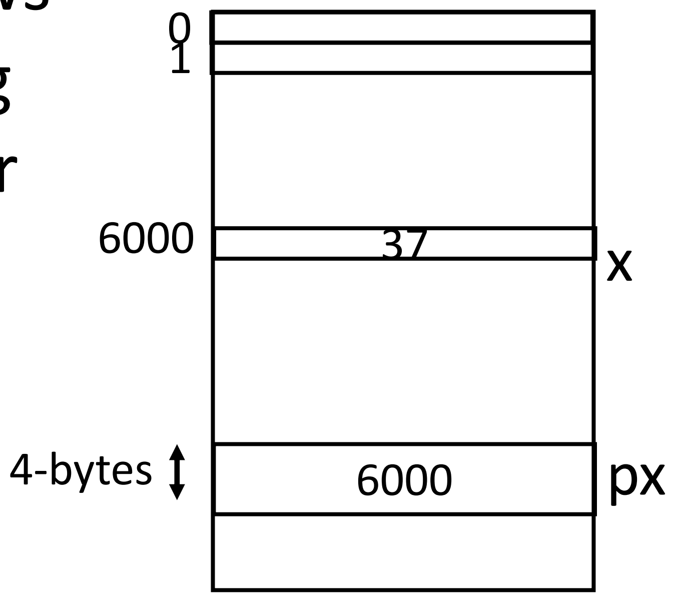
Pointers – The dereference operator *

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

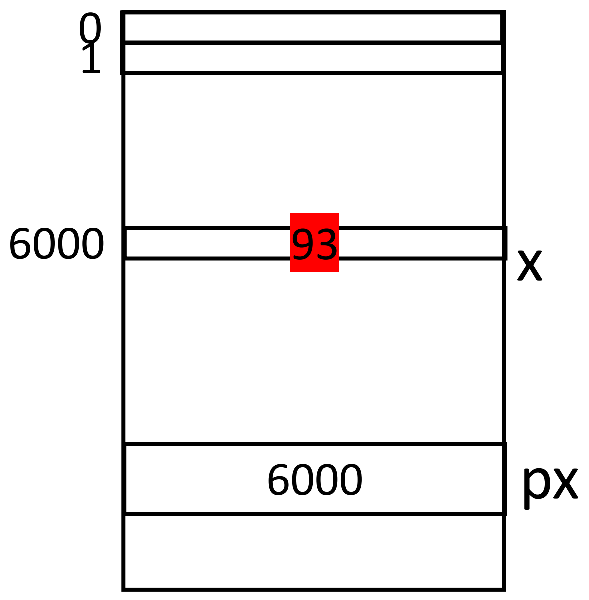
```
*px = 93;
```

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;



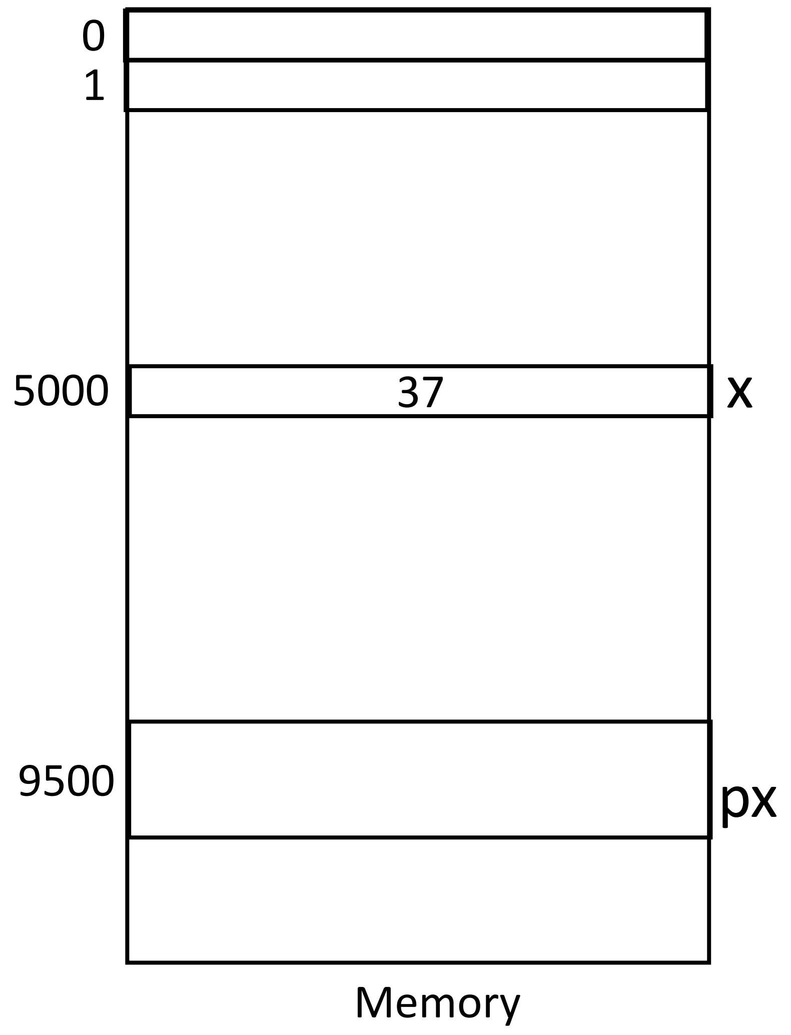
Memory after the statement *px=93;

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 - 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;
}
```

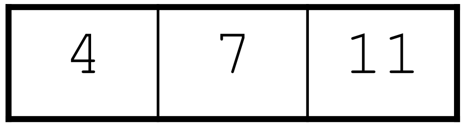
Pointers – example 2

```
// 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;
}
```

Arrays and pointers

- The array name holds the starting address of the array

```
int vals[] = {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;
```

valid

```
px = x;
```

Not valid

~~`x = px;`~~

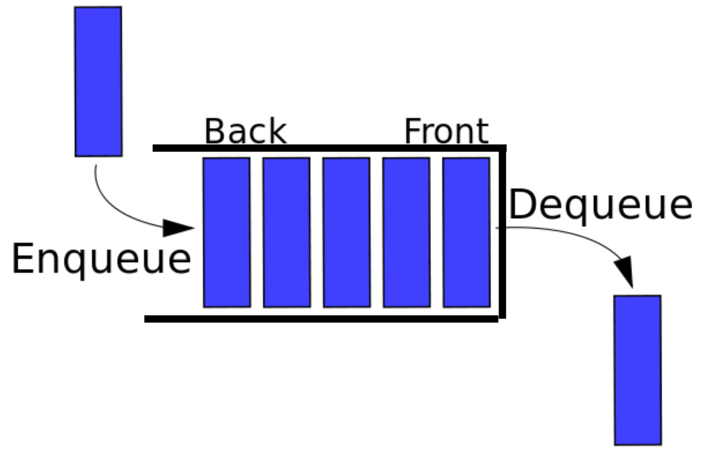
- An array declaration allocates memory for the number of elements inside the array, whereas the declaration of a pointer allocates only the memory required to hold an address.

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;
}
```

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



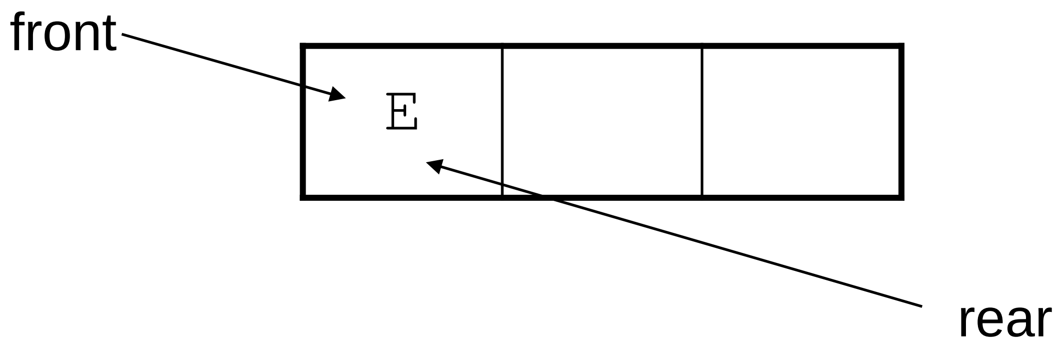
The Queue data structure - operations

- 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.

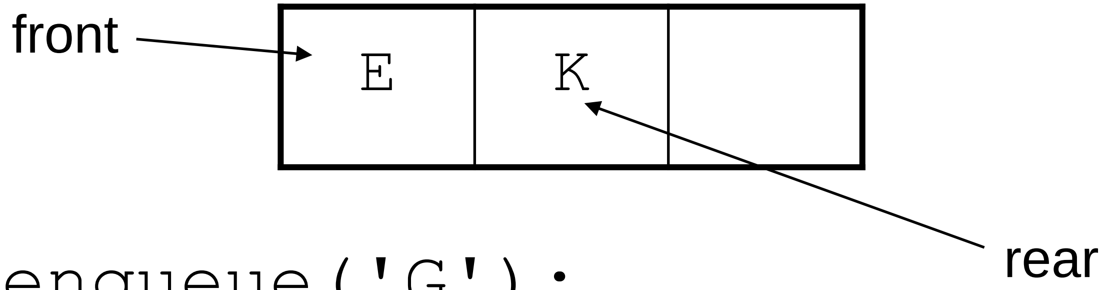
- A currently empty queue that can hold `char` values:

- `enqueue('E');`

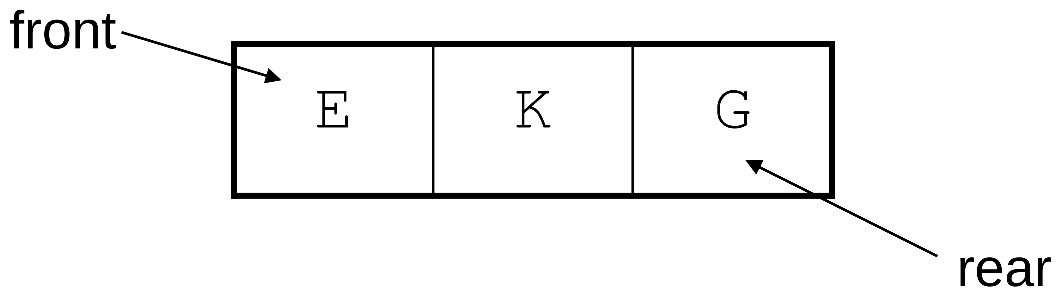


Queue operations – cont.

- `enqueue('K');`

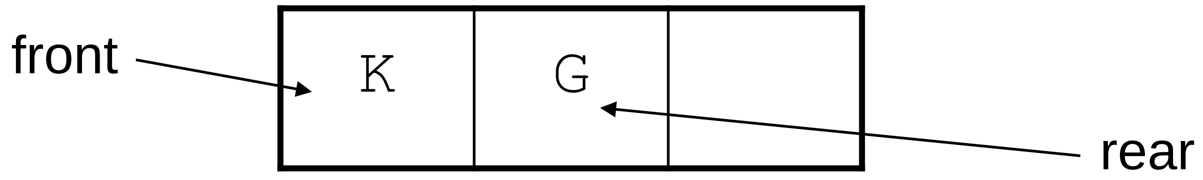


- `enqueue('G');`

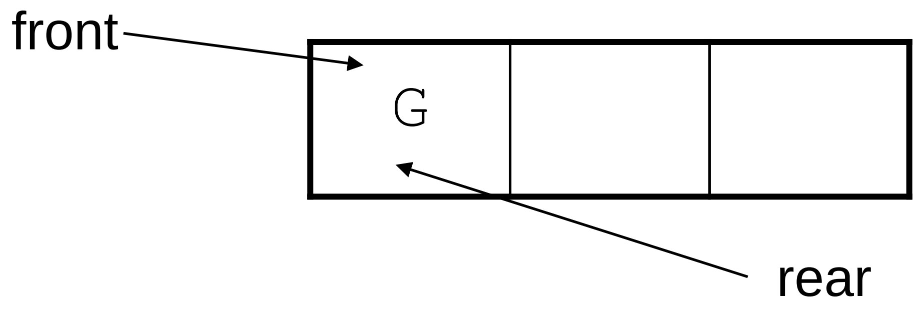


Queue operations – cont.

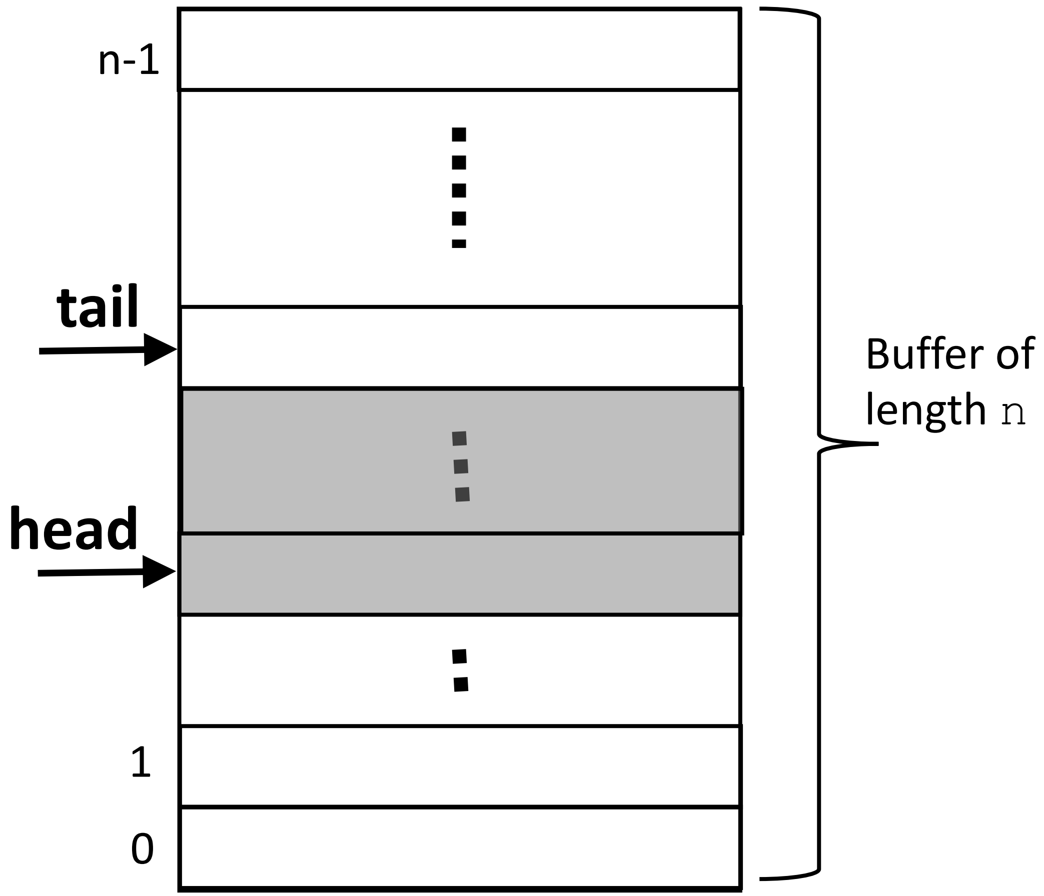
- `dequeue(); // remove E`

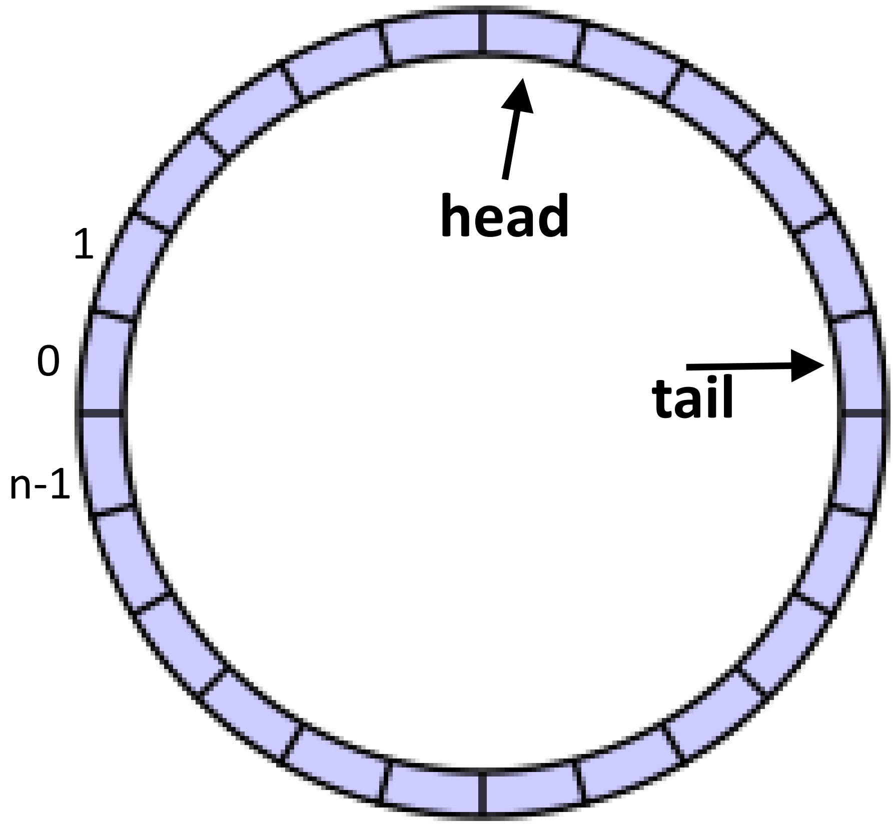


- `dequeue(); // remove K`



The Queue data structure – Buffer Implementations





The queue data structure - Design/algorithms

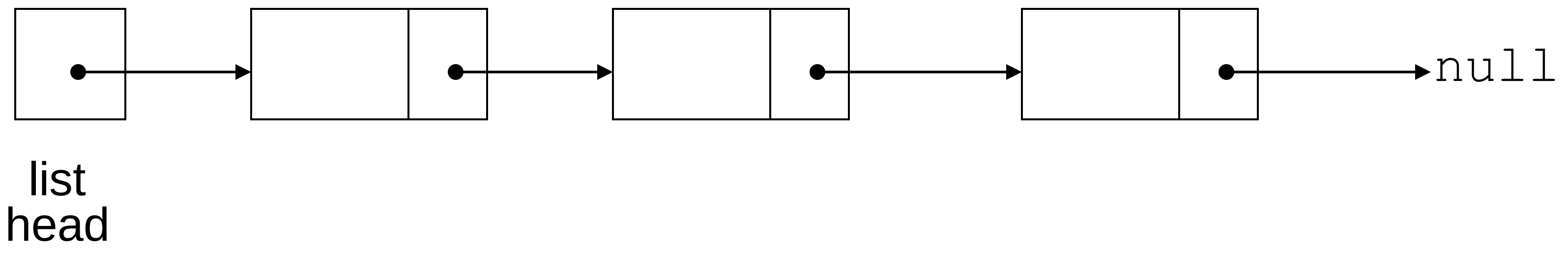
- Create a buffer of length n
- Create some variables : (all initialized to zero)
  - Use a variable (tail or write_index) to hold the index where new data should be written.
  - Use a variable (head or read_index) to point to the array index from which data may be read.
  - A count variable to hold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

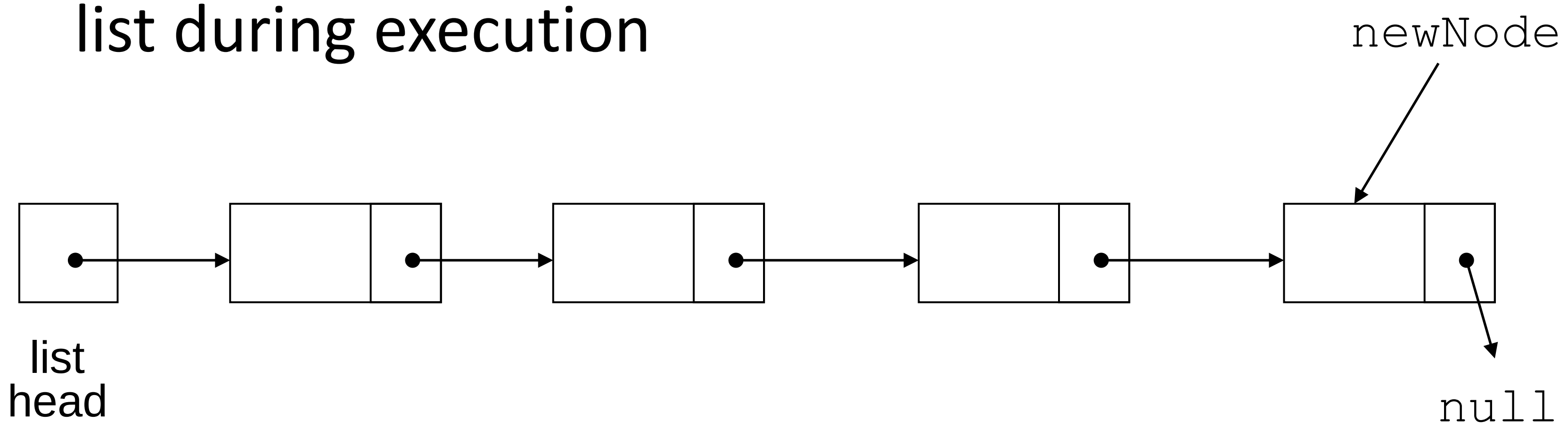
Introduction to the Linked Lists

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



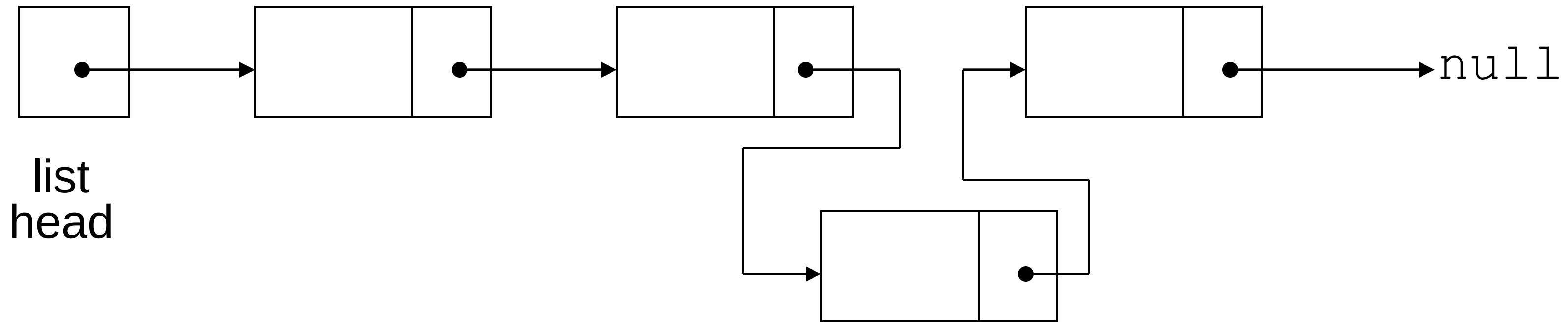
Introduction to the Linked Lists - cont.

- 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

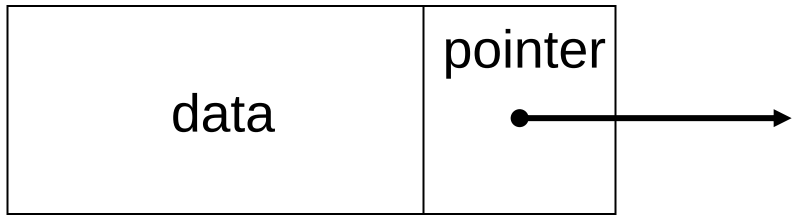
- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size
- Linked lists can insert a node between other nodes easily



Node Organization

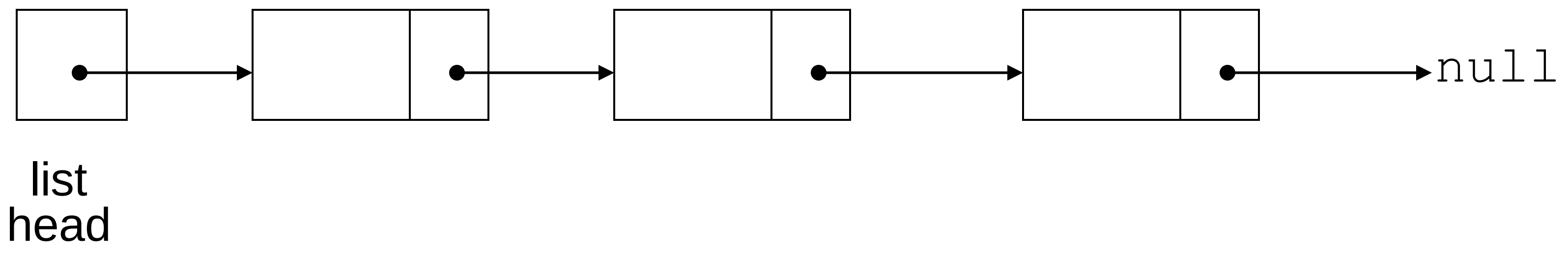
- A node contains:
  - **data**: one or more data fields – may be organized as structure, object, etc.
  - **a pointer**: that can point to another node

  - 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

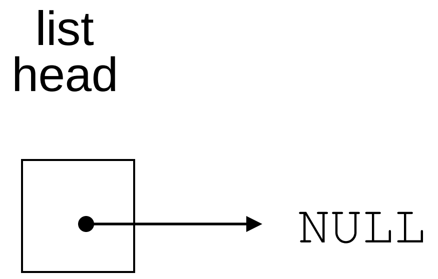
- Linked list contains 0 or more nodes:



- 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 empty list
- In this case the list head points to `null`



Declaring a Node

- Declare a node:

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

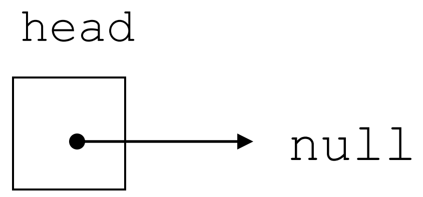
- No memory is allocated at this time

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

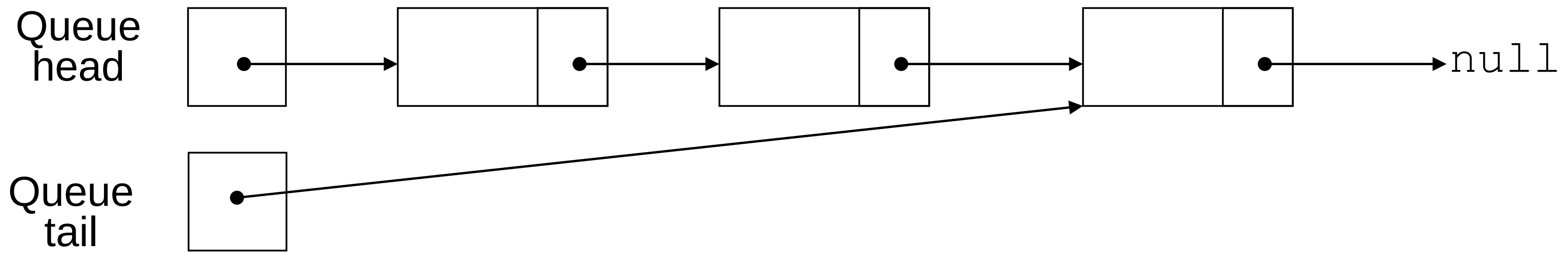
- Is used to indicate end-of-list (null = address 0 in memory which normally not a valid address)
- Should always be tested for before using a pointer:

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



Chapter 3: Processes

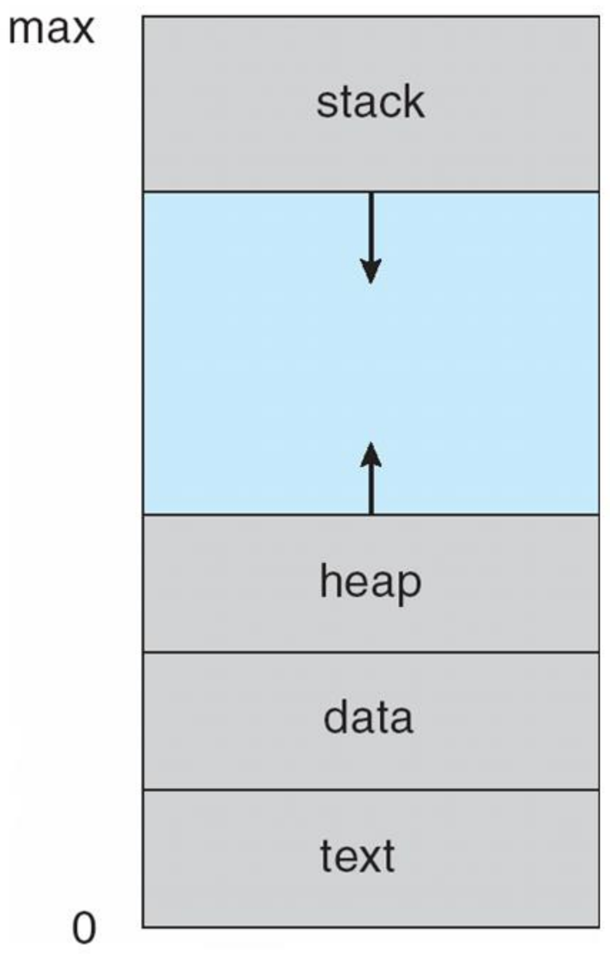
- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

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*** and ***process*** almost interchangeably
  - In traditional operating sytems (i.e. those running workstations or servers): process = job = task
- **Process** – a program in execution; process execution generally 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** containing temporary data
    - Function parameters
    - Saved CPU registers (including return addresses)
    - Local variables
  - **Heap** containing memory dynamically allocated during run time. Grows opposite to the stack
- A process also occupies/uses CPU registers (not shown on Fig.).

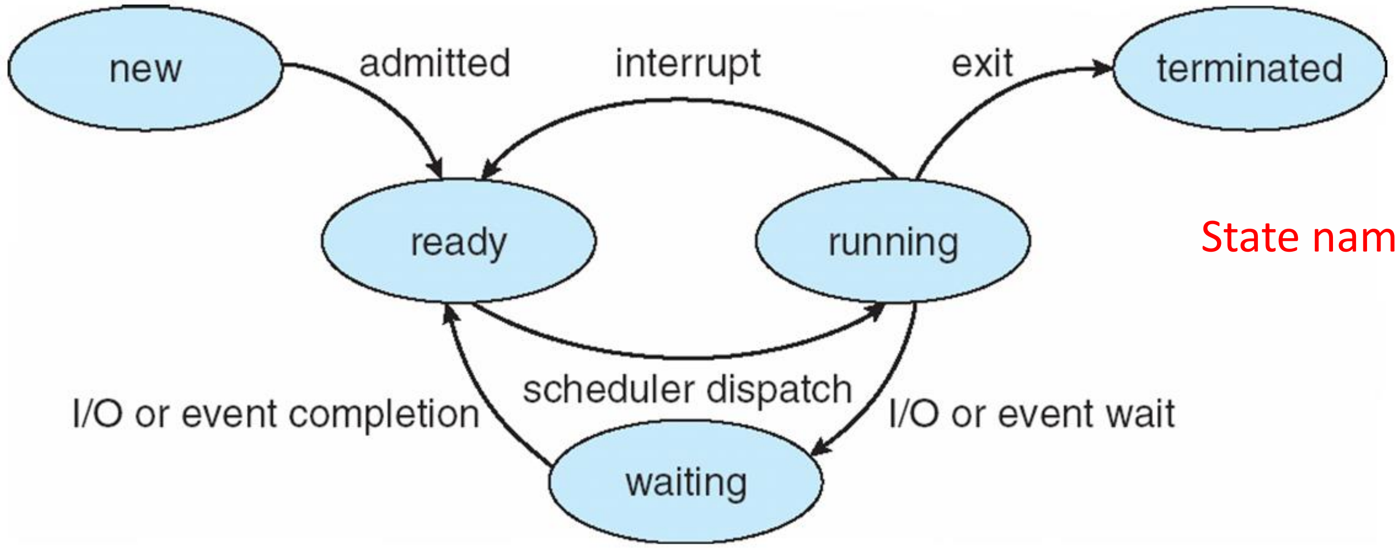


The Process Concept (Cont.)

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when its executable file is loaded into main memory and is regsstered 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



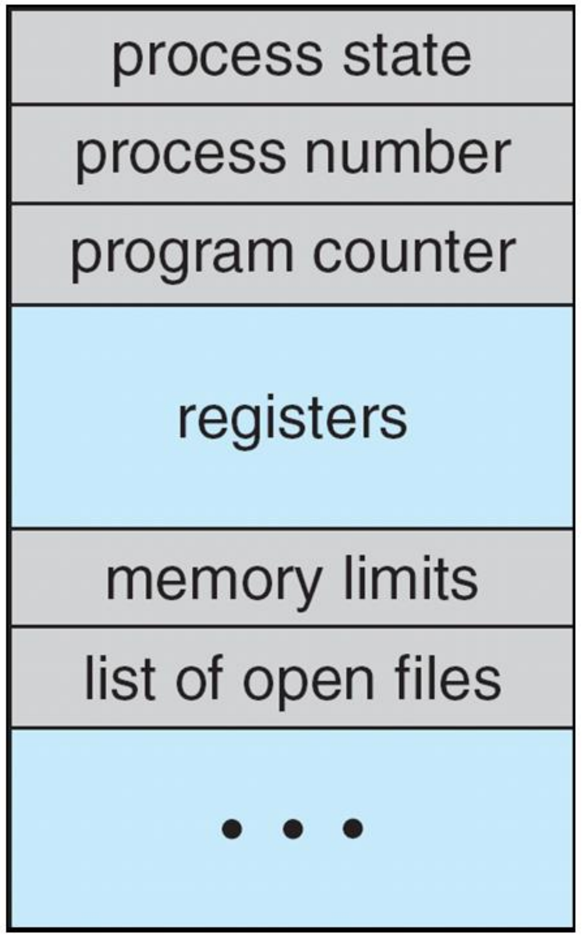
State names are generic

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



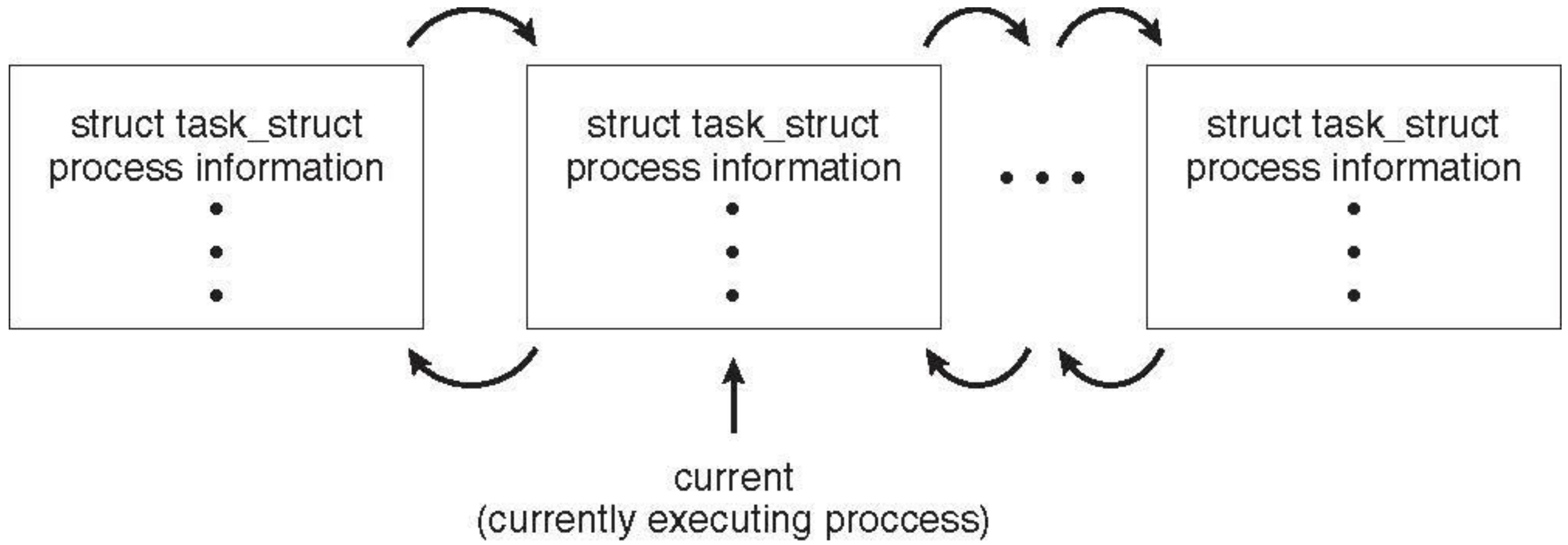
Threads

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

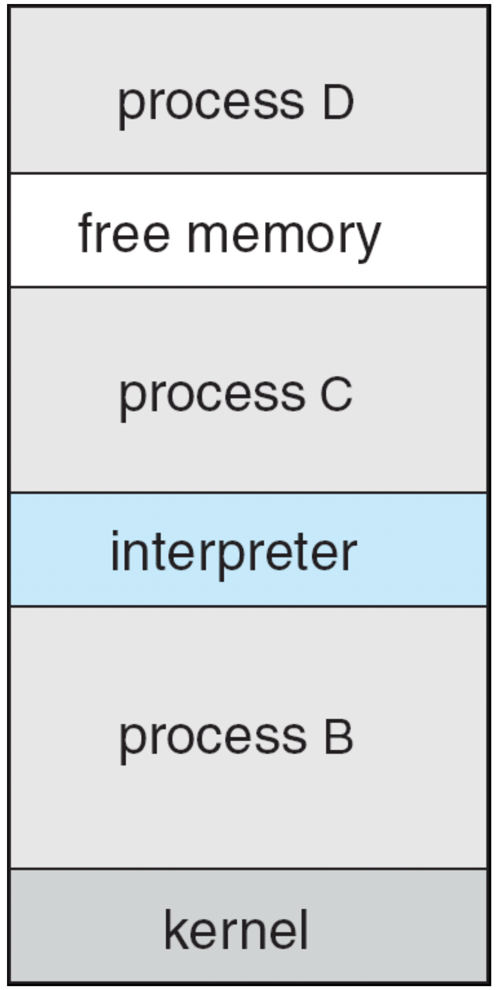
Process Representation in Linux

Represented by the C structure `task_struct`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process’s parent */
struct list_head children; /* this process’s children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



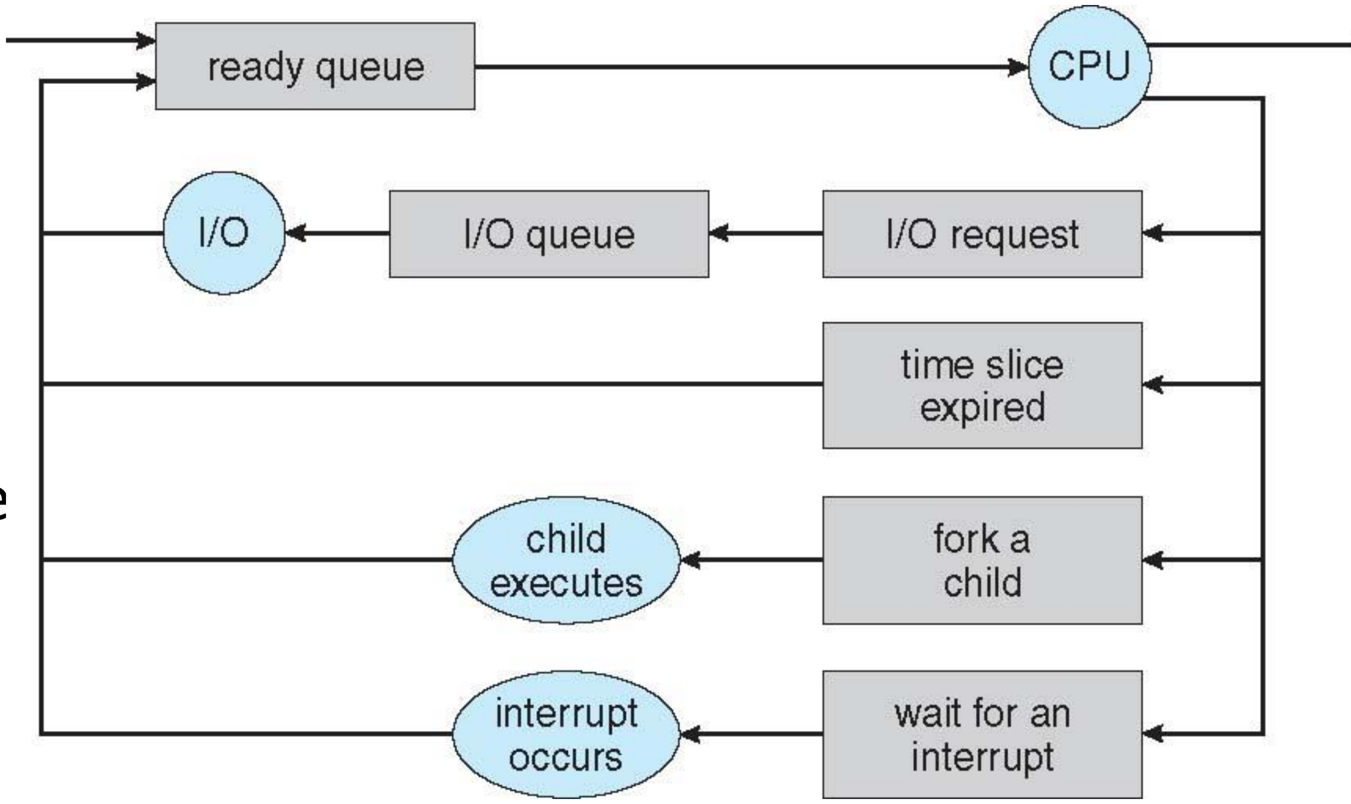
Where is the PCB for process C ?



main memory

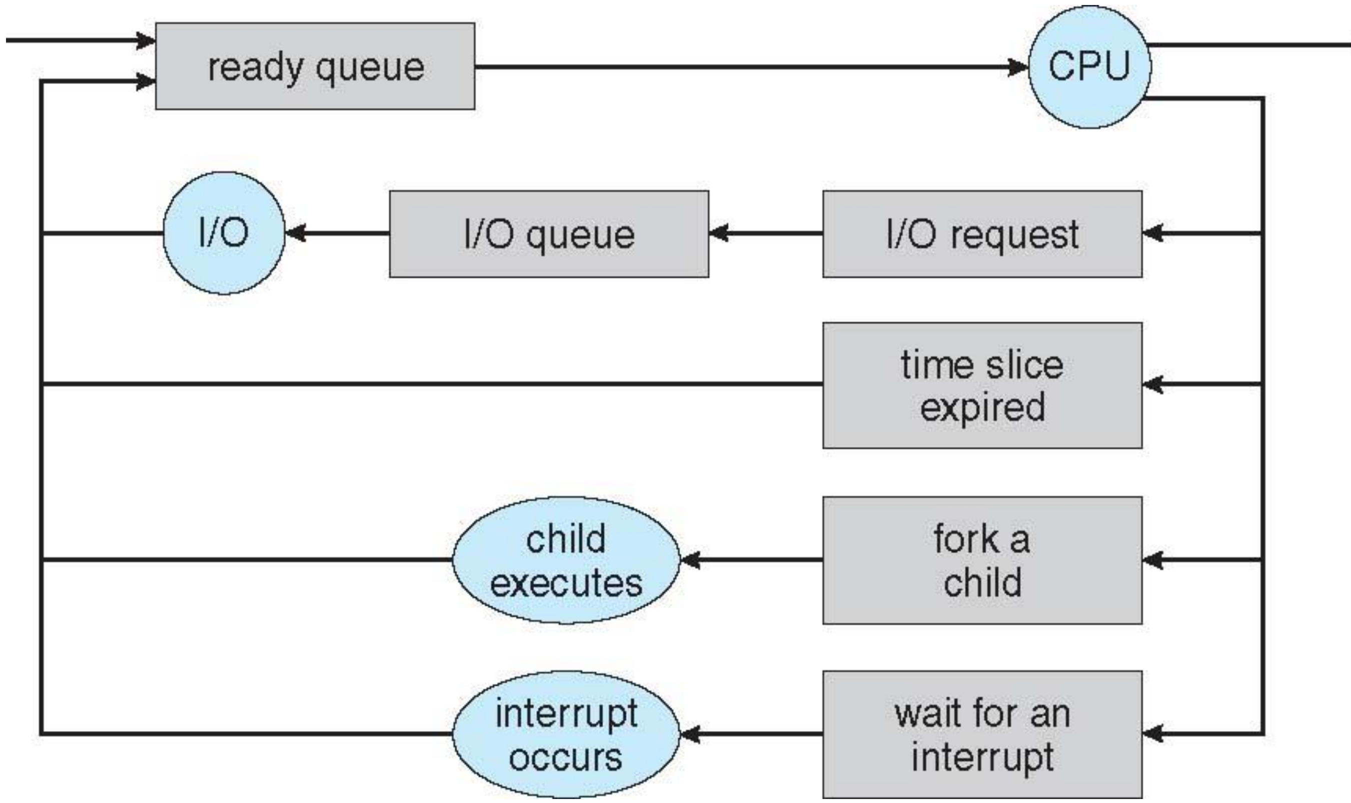
3.2 Process Scheduling

- In a single CPU system, the scheduler chooses only one process to run at a time, while the rest of available processes must wait till the CPU is free.
- The **kernel** must maximize CPU use and quickly switch processes onto the CPU for time sharing
- **Process scheduler** is a function (within a module) that selects among "ready" processes for next execution on CPU



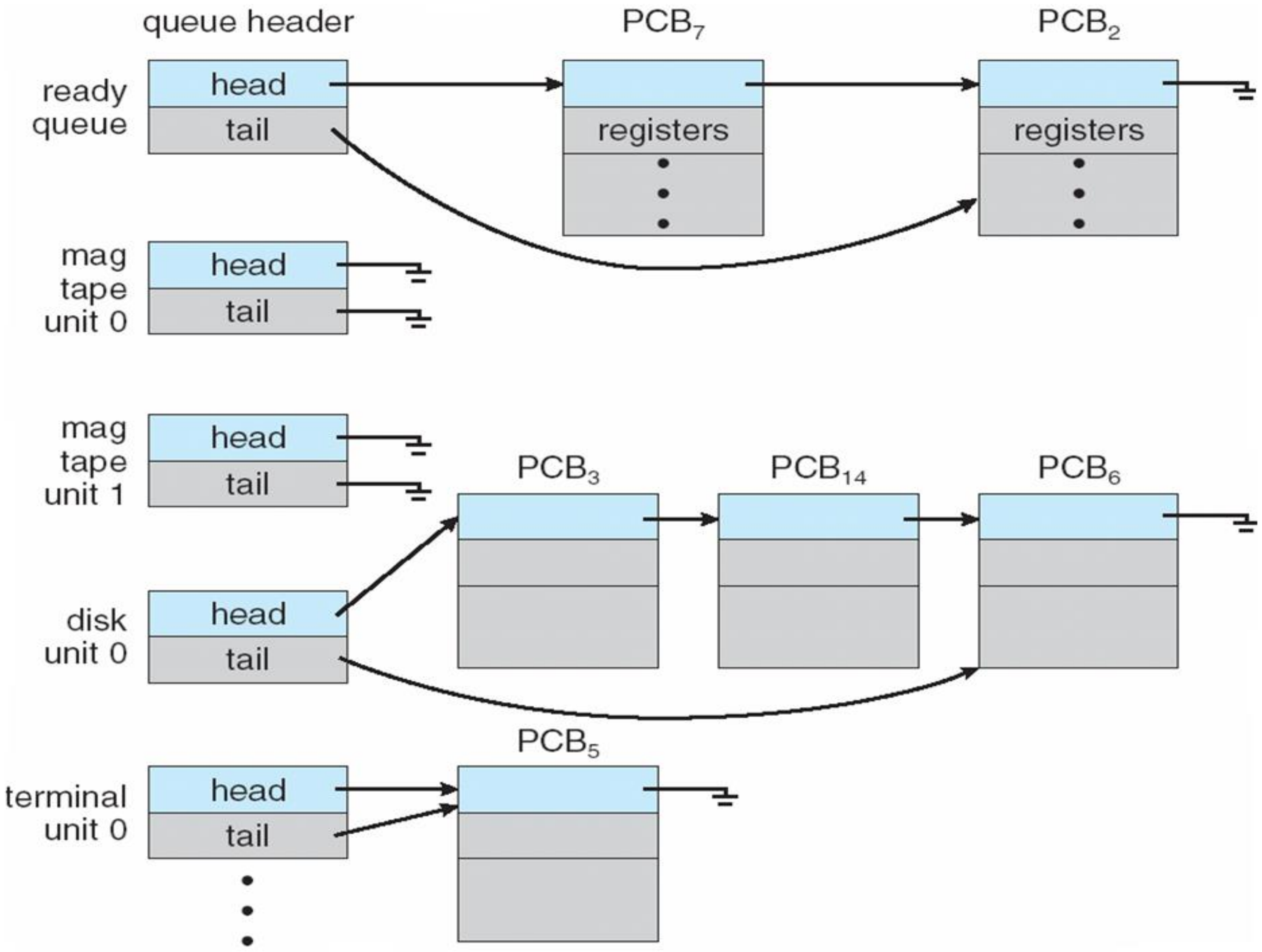
Queueing diagram represents queues, resources, flows

- **Process manager** maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute (stored as a linked list)
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues



Queueing diagram represents queues, resources, flows

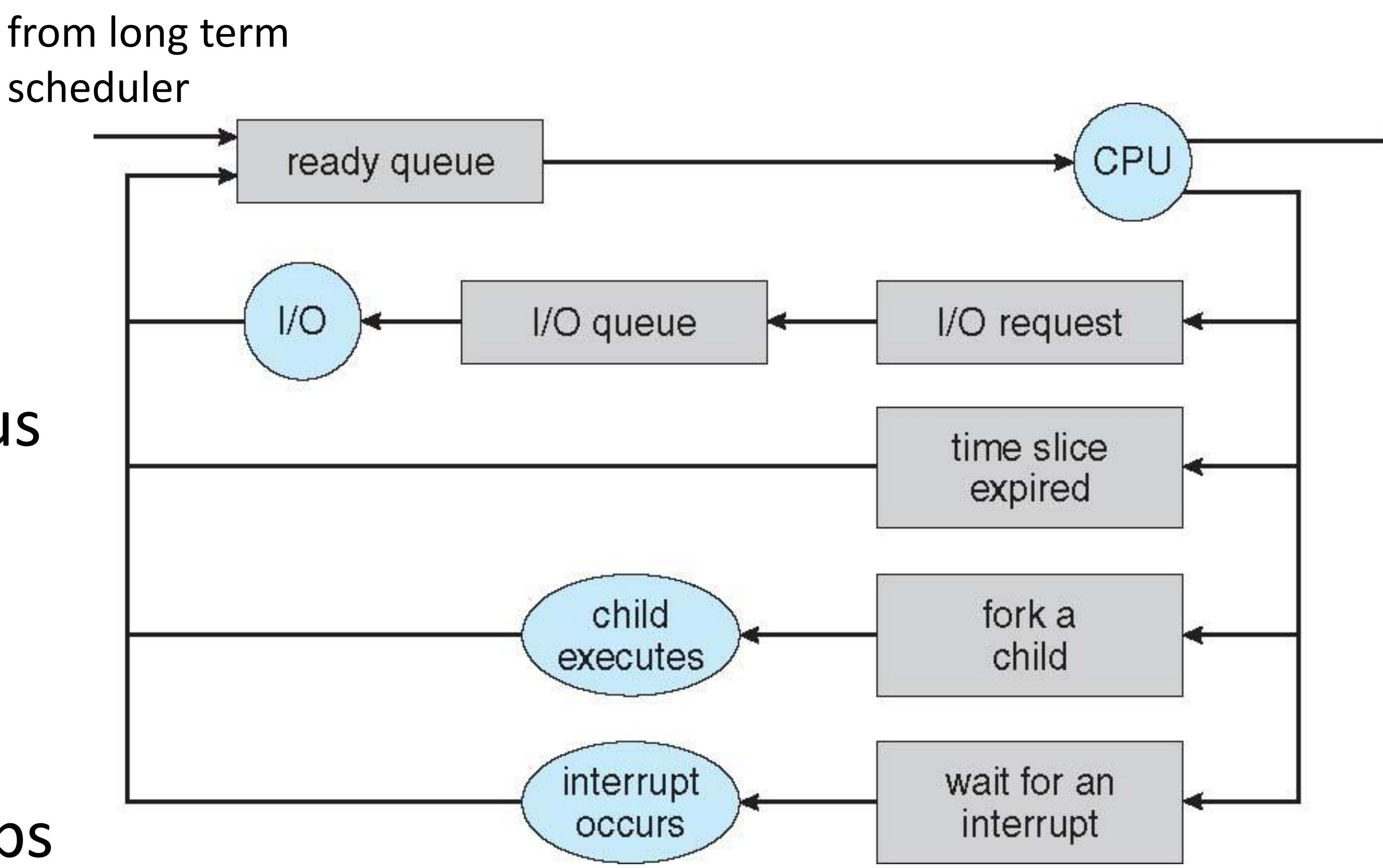
Ready Queue And Various I/O Device Queues



Schedulers

Long-term scheduler

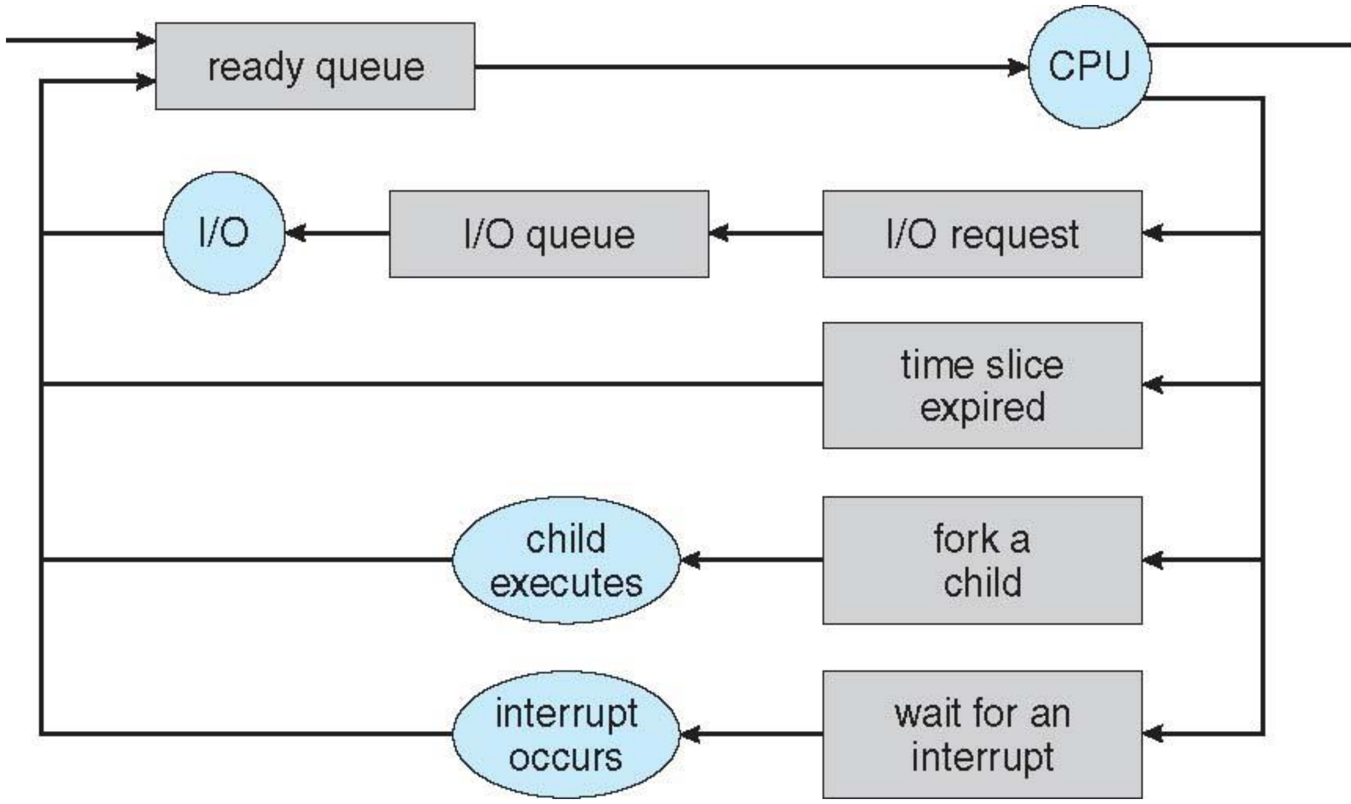
- Occurs in batch systems
- Invoked when a process exits, i.e. infrequently, in seconds or minutes $\Rightarrow$ (thus is allowed to be slow)
- Selects which processes should be admitted, i.e. brought into the ready queue, from the pool of jobs waiting.
- Controls the **degree of multiprogramming**



Queueing diagram represents queues, resources, flows

Schedulers – cont.

- **Short-term scheduler** (or just **scheduler**) – selects which process should be executed next and allocates a CPU for it:
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)



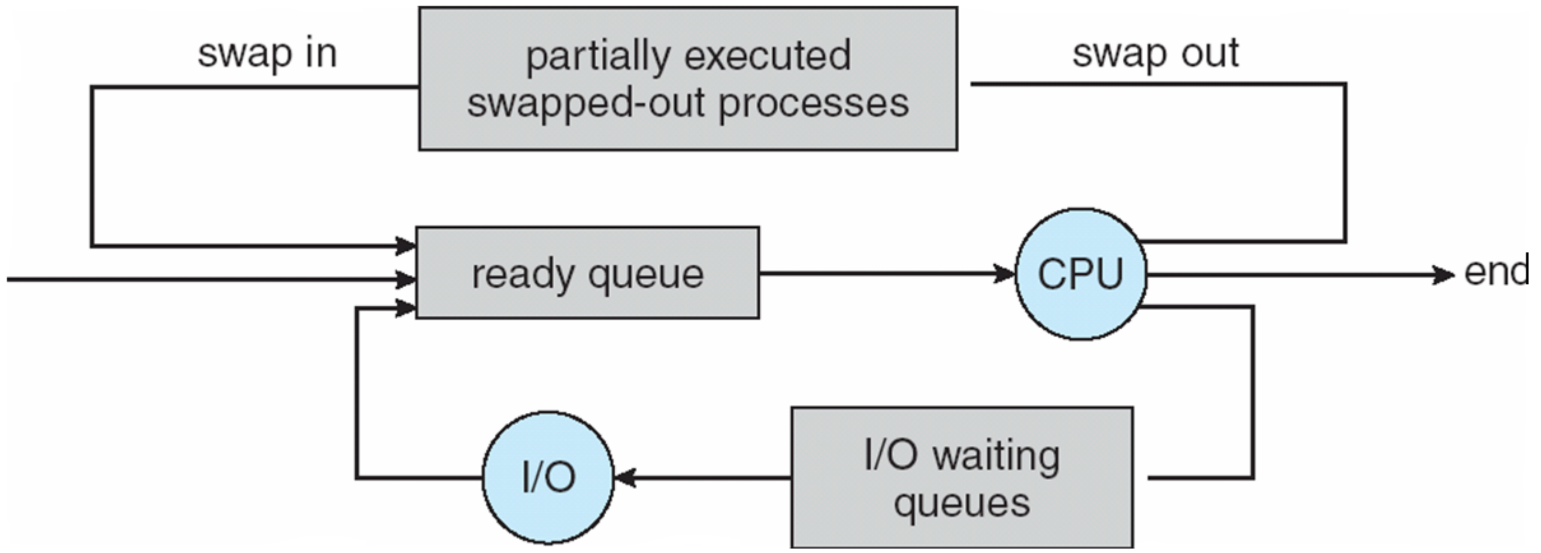
Queueing diagram represents queues, resources, flows

Schedulers – cont.

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix*** of I/O-bound and CPU-bound processes.
- Unix and windows do not implement long term scheduling, and thus their stability (which relies on degree of multi-programming) relies on the user’s behavior, i.e. the user won’t try to run more processes if the system is slowing down and becoming irresponsive.

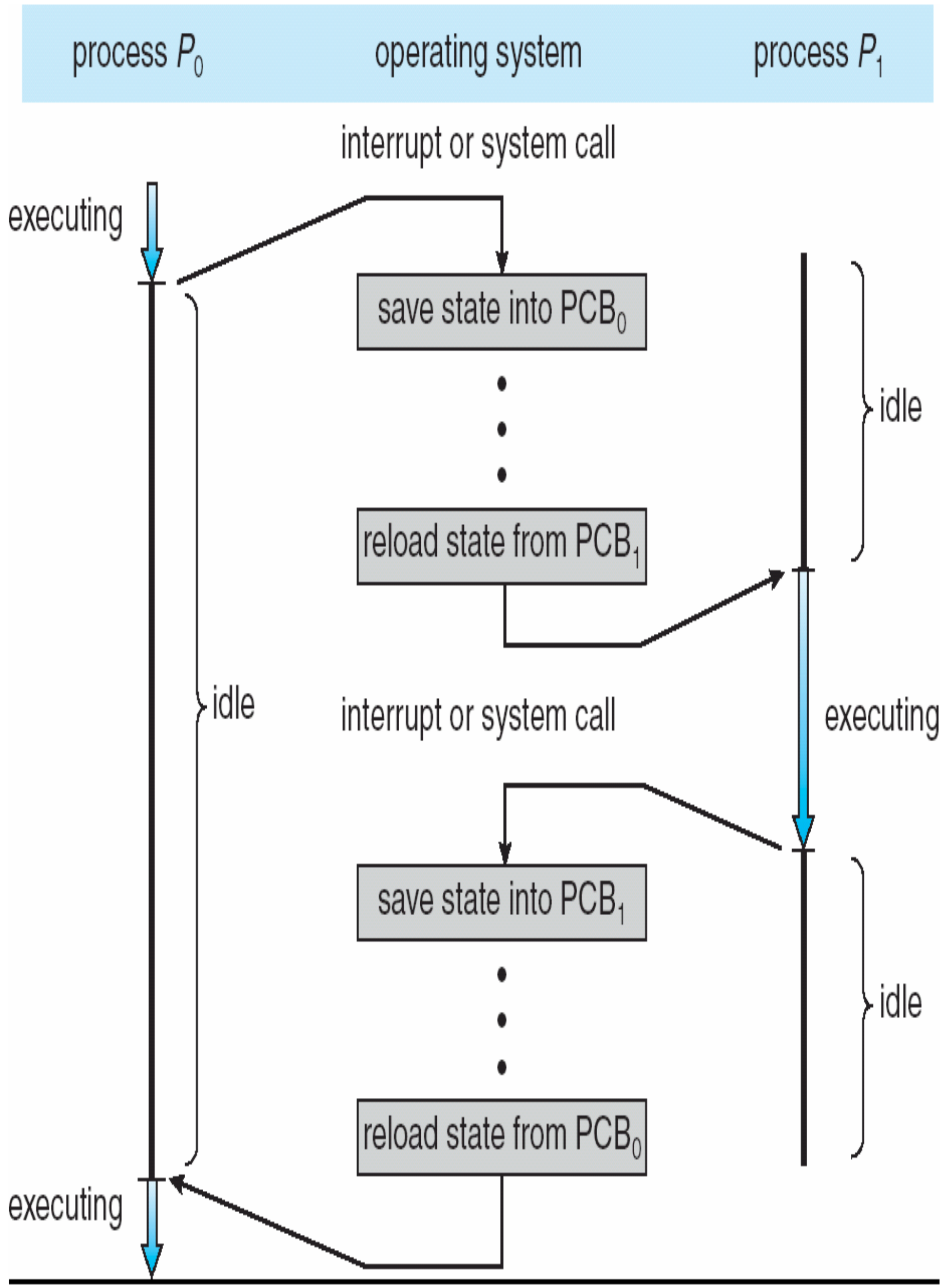
Schedulers – cont.

- **Medium-term scheduler (aka swapping scheduler)** can be added if degree of multiple programming needs to decrease, or if the process mix (I/O-bound vs CPU-bound) needs to be adapted.
    - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- The **context** of a process is represented in the PCB.
- It involves a state save, then a state restore.



Context Switch

- Context-switch time is **overhead**; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
  - The address space needs to be preserved as part of the PCB. How it is preserved and what amount of work is needed depends on the memory management method of the OS.