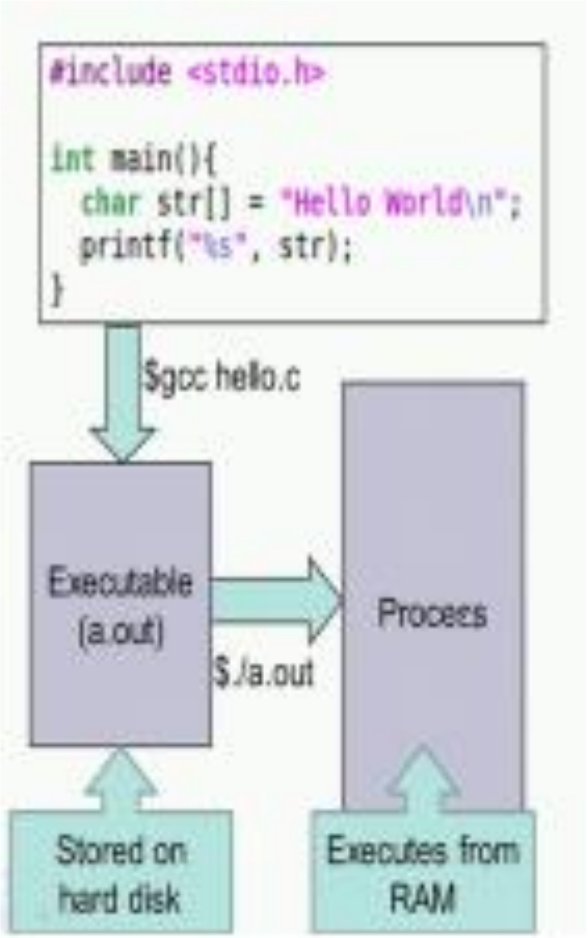
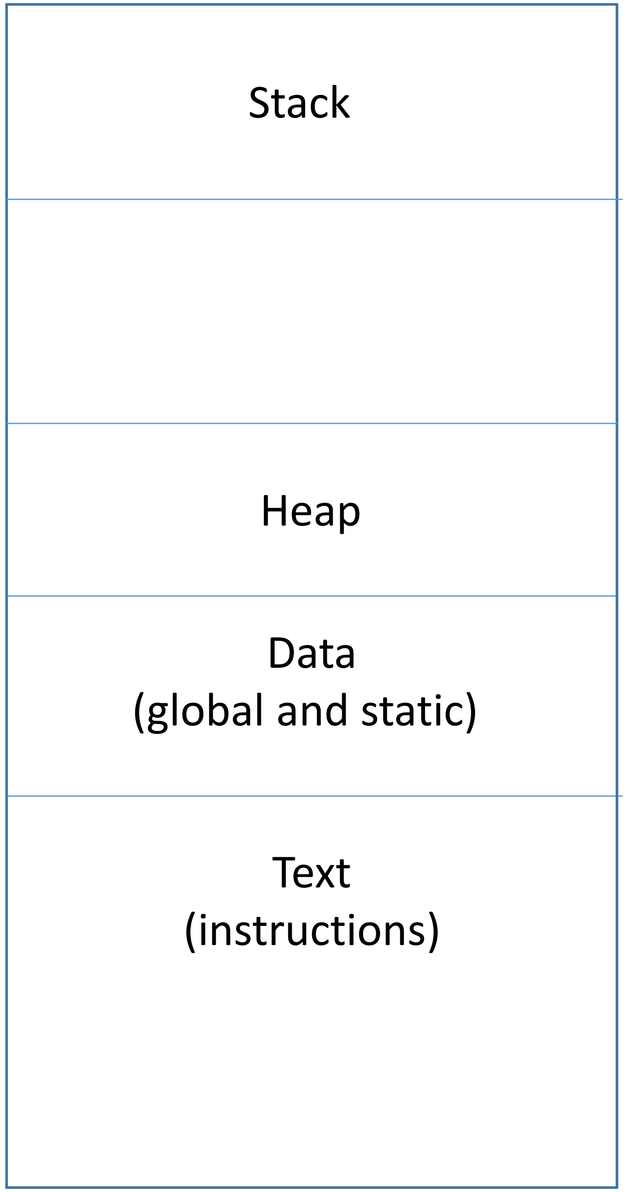
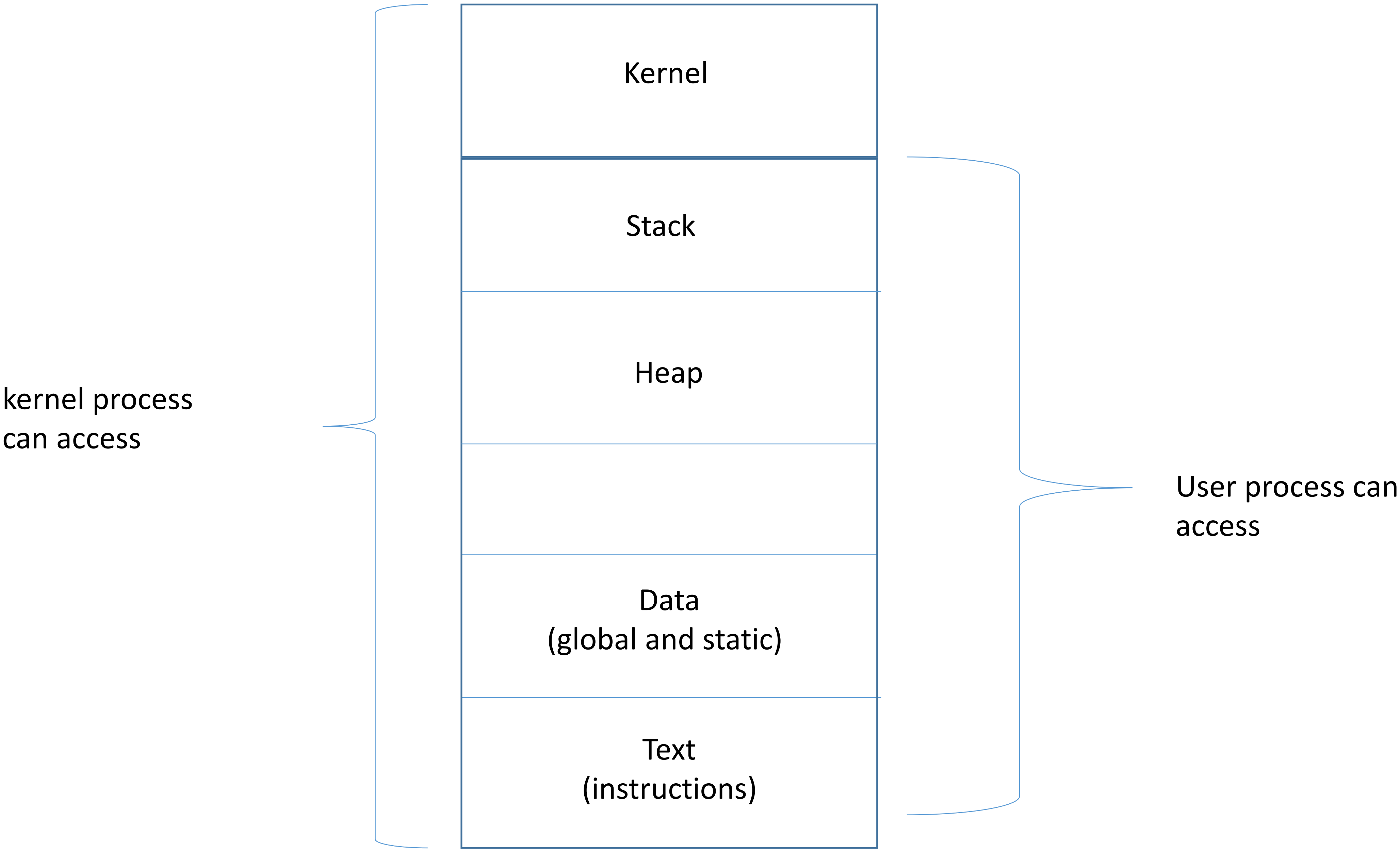
Processes

- From programs to processes
  - Source code (.c)
  - Executable (./a.out)
  - Process (gets loaded into RAM)
    - Executable instructions
    - Stack
    - Heap
    - State in the OS (registers, list of open files, other related processes, etc.)

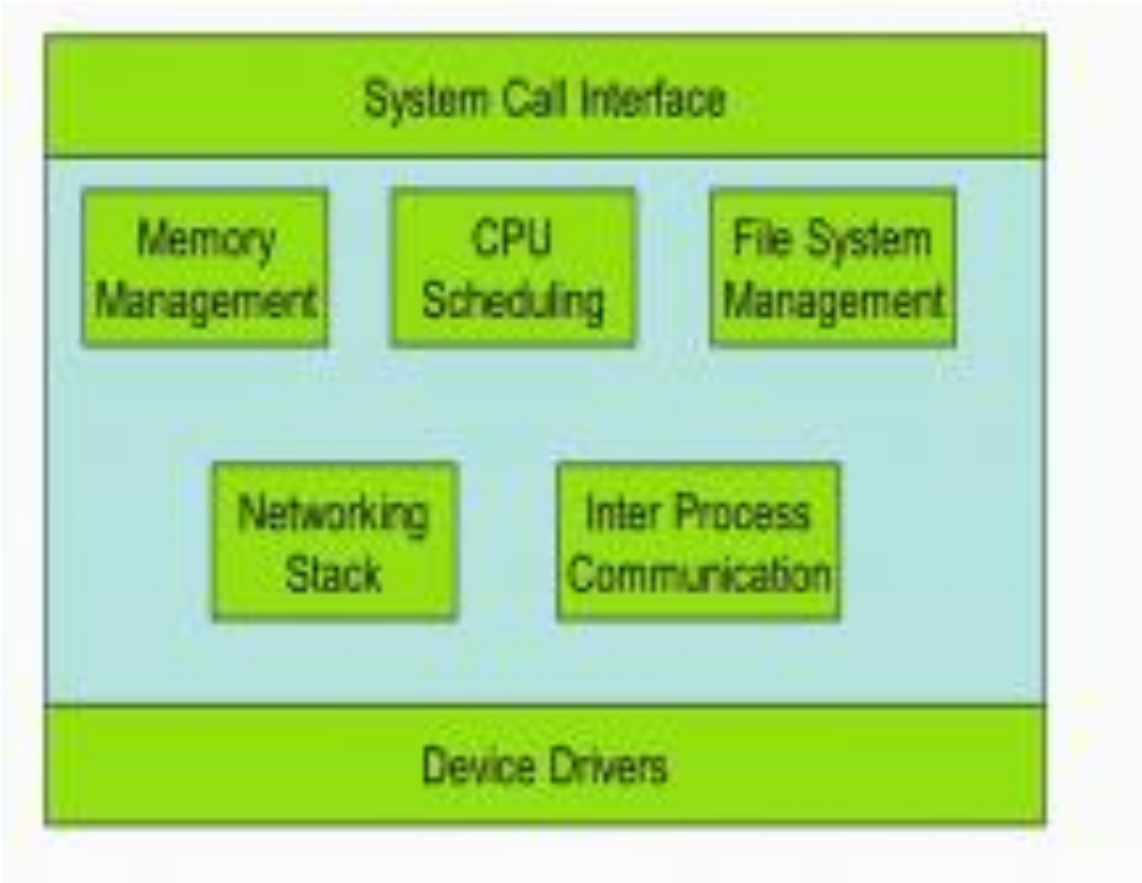


```
#include<stdio.h>
#include<stdlib.h>
int count;
int main()
{
   int n, *ptr;
   scanf(“%d”,&n);
   ptr = (int*) malloc(n*size(int));
   *ptr = 1;
   fact(n,ptr);
   printf(“Factorial is: %d”, *ptr);
   free(ptr);
}
void fact(int num, int *fact)
{
   count++;
   if (num==1) return;
   *fact = *fact * num;
   fact(num-1, fact);
}
```




kernel process
can access
Kernel
Stack
Heap
Data
(global and static)
Text
(instructions)
User process can
access

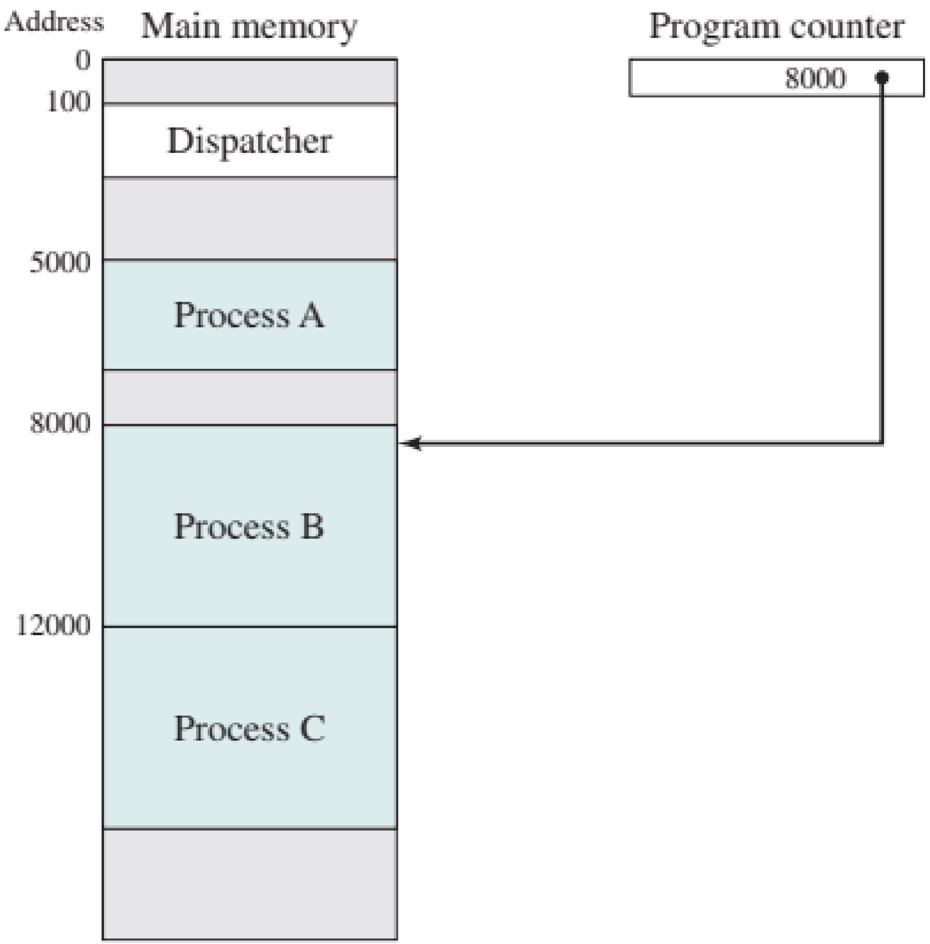
OS structure



- Principal responsibility
  - Controlling the execution of processes
  - Includes the pattern for execution
  - Allocating resources for processes

Process states

Traces of the processes A, B, and C
Trace: sequence of instructions that execute for a process



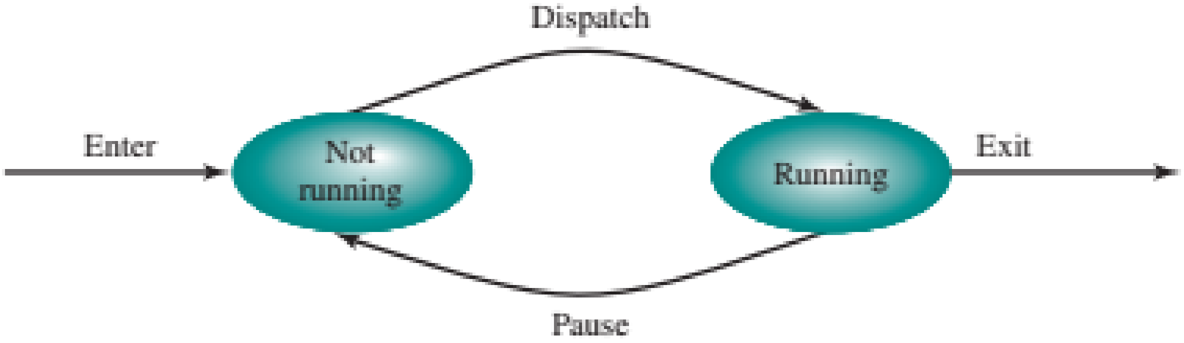
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of process A (b) Trace of process B (c) Trace of process C

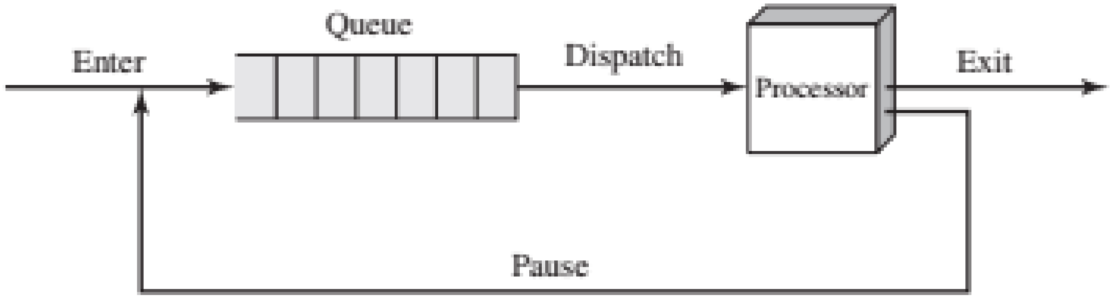
1	5000
2	5001
3	5002
4	5003
5	5004
6	5005
----------------------Time-out	
7	100
8	101
9	102
10	]103
11	]104
12	105
13	8000
14	8001
15	8002
16	8003
----------------------I/O request	
17	100
18	101
19	102
20	103
21	104
22	105
23	12000
24	12001
25	12002
26	12003
27	12004
28	12005
----------------------Time-out	
29	100
30	101
31	102
32	103
33	104
34	105
35	5006
36	5007
37	5008
38	5009
39	5010
40	5011
----------------------Time-out	
41	100
42	101
43	102
44	103
45	104
46	105
47	12006
48	12007
49	12008
50	12009
51	12010
52	12011
----------------------Time-out	

- 52 instruction cycles
- Trace from the processor's POV
- Assumption: OS allows a process to continue for only six instruction cycles

Two state process model

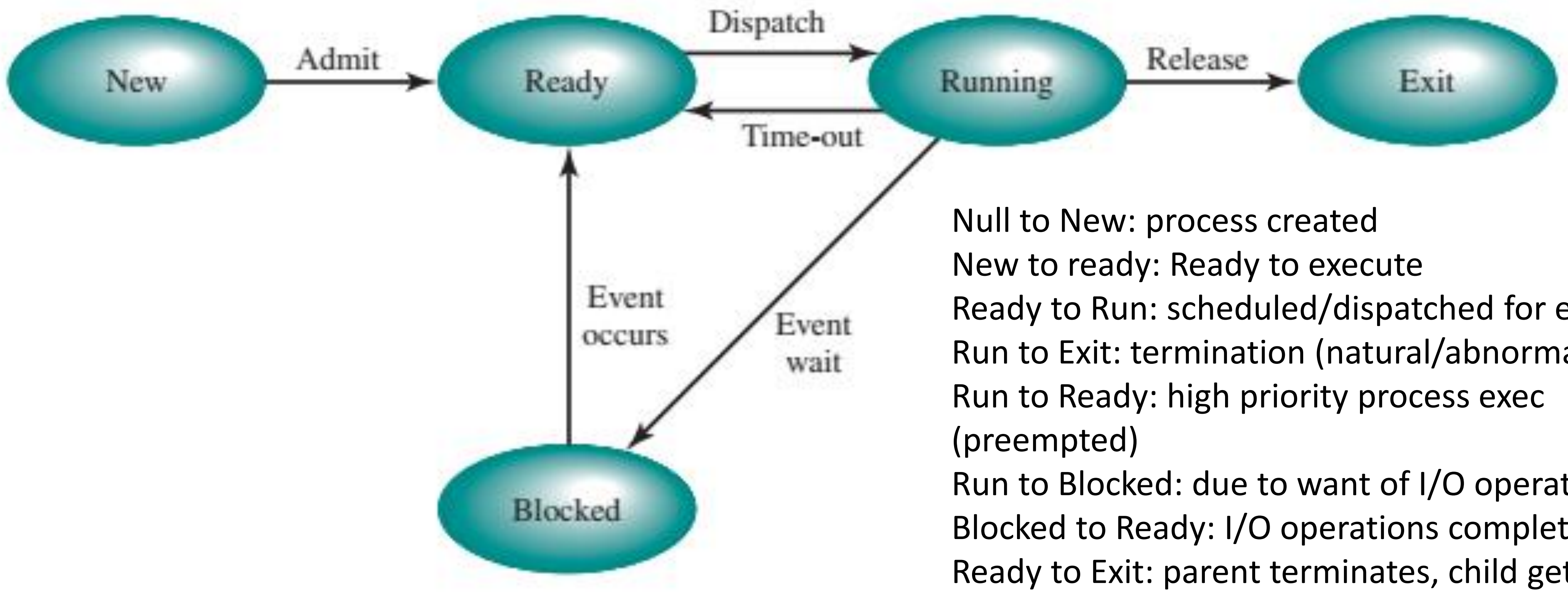


(a) State transition diagram



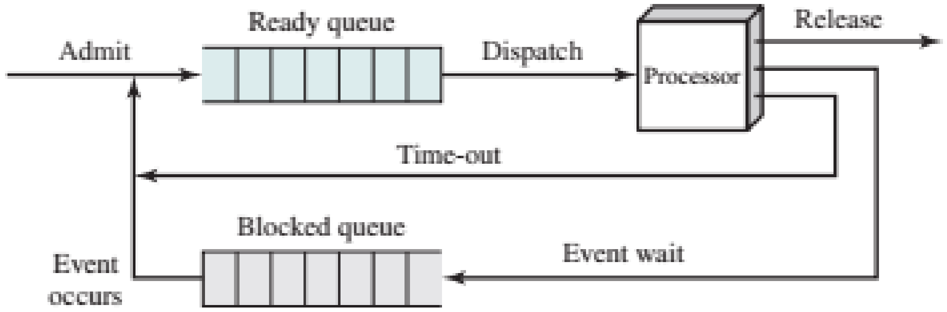
(b) Queueing diagram

Five state model

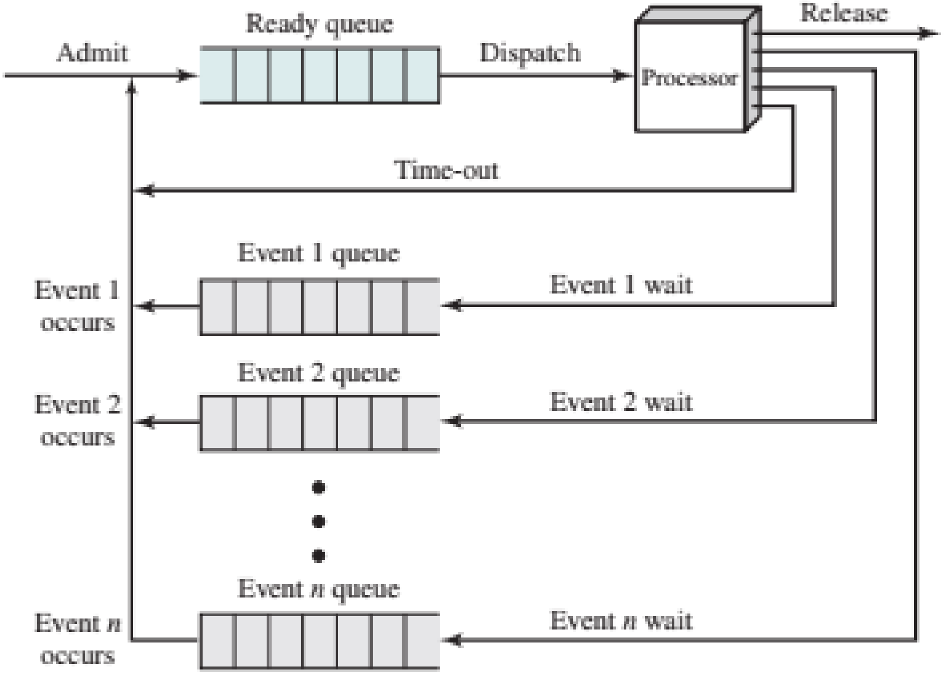


Null to New: process created
New to ready: Ready to execute
Ready to Run: scheduled/dispatched for exec
Run to Exit: termination (natural/abnormal)
Run to Ready: high priority process exec (preempted)
Run to Blocked: due to want of I/O operation
Blocked to Ready: I/O operations completed
Ready to Exit: parent terminates, child gets terminated
Blocked to Exit: parent terminates, child gets terminated

Queues

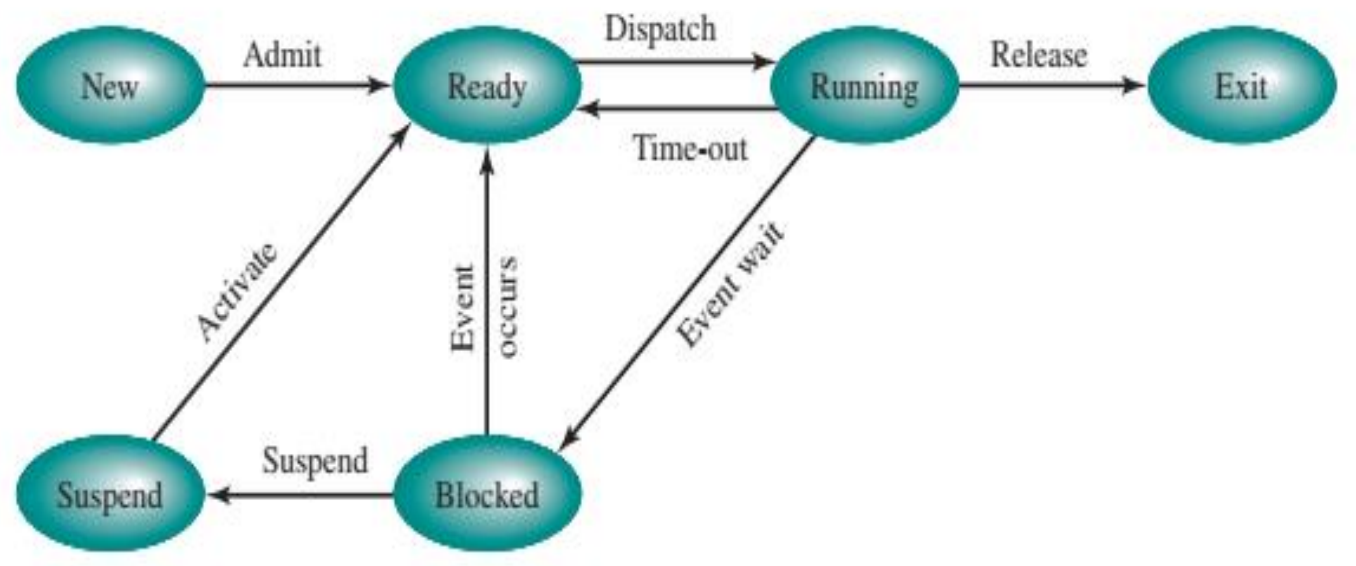


(a) Single blocked queue



(b) Multiple blocked queues

Suspend State



All the processes in blocked state are still in main memory

Still processor can sit idle.

Solution?

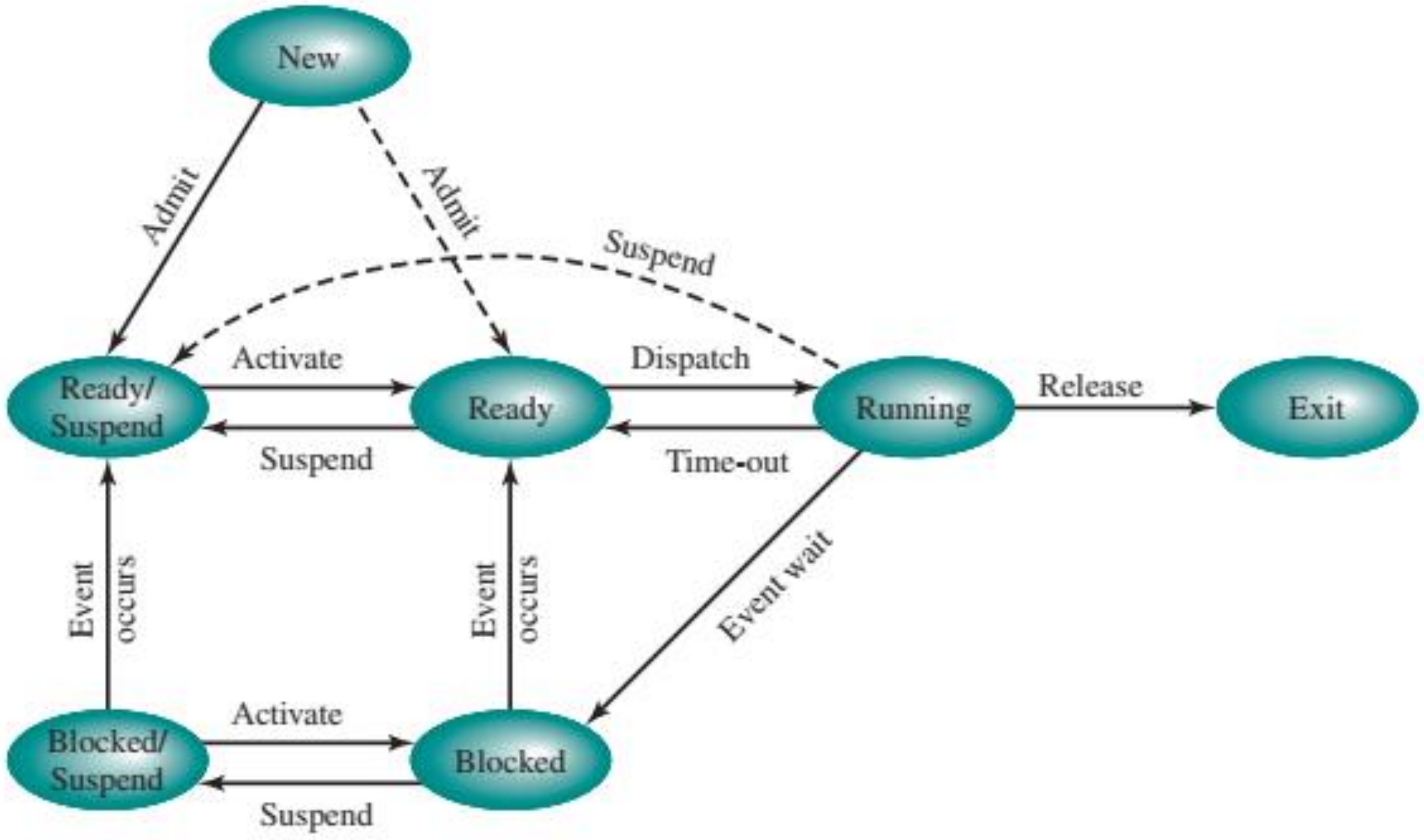
1) Increase the size of main memory
2) Employ swapping mechanism
   Main memory to disk

Swapping is in itself an I/O operation; disk I/O is better than printer I/O or Keyboard I/O

Ready: process in main memory, ready for execution

Blocked: process in main memory, waiting for an event

Suspend: process in secondary memory, waiting for an event

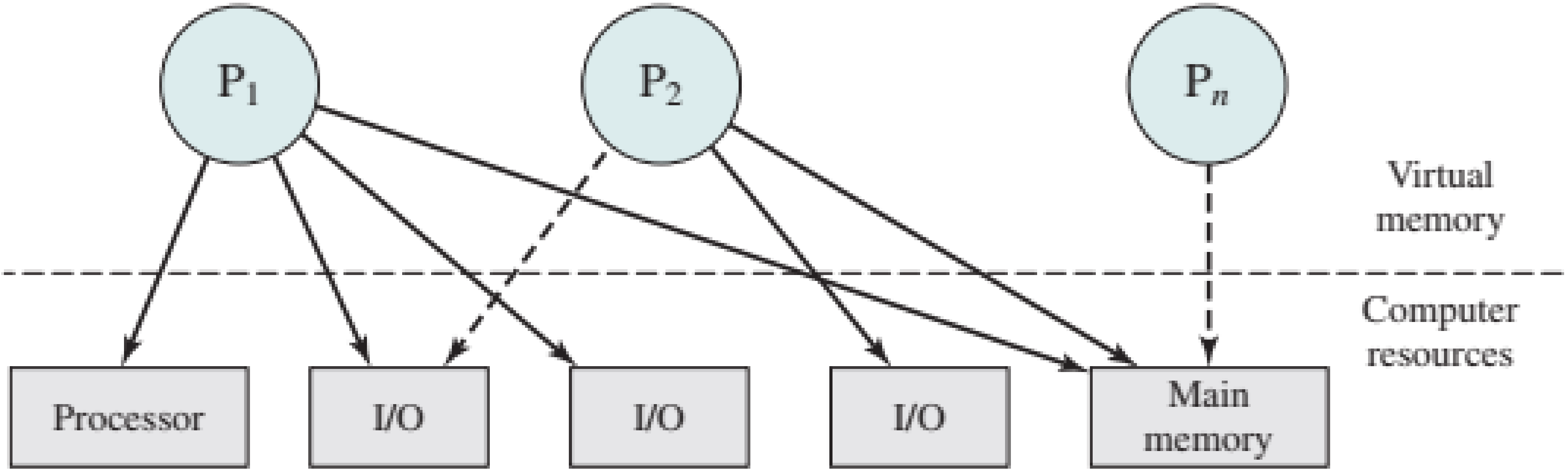


Ready: process in main memory, ready for execution

Blocked/suspend: process in main memory, waiting for an event

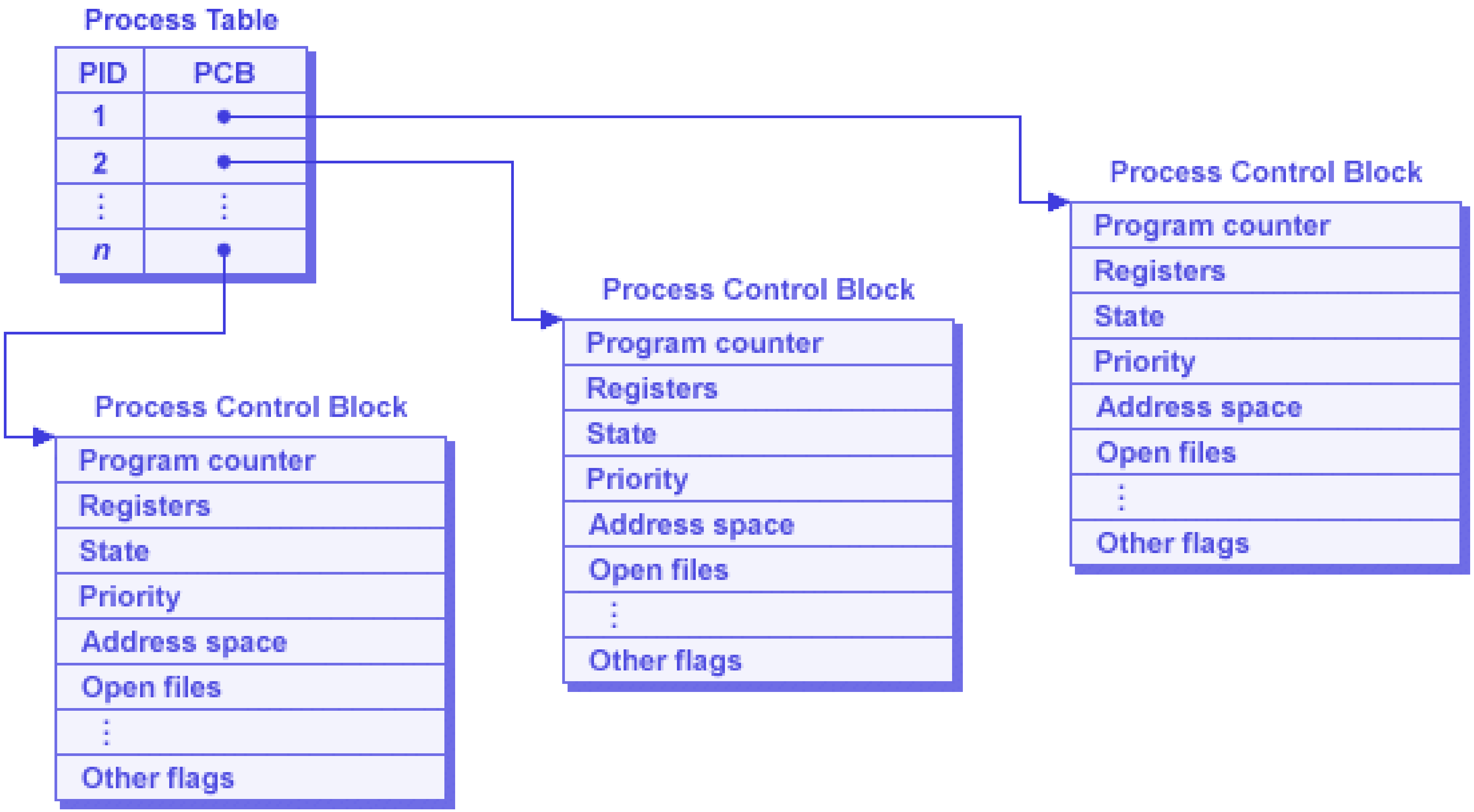
Ready/suspend: process in secondary memory, waiting for an event

Processes and Resources

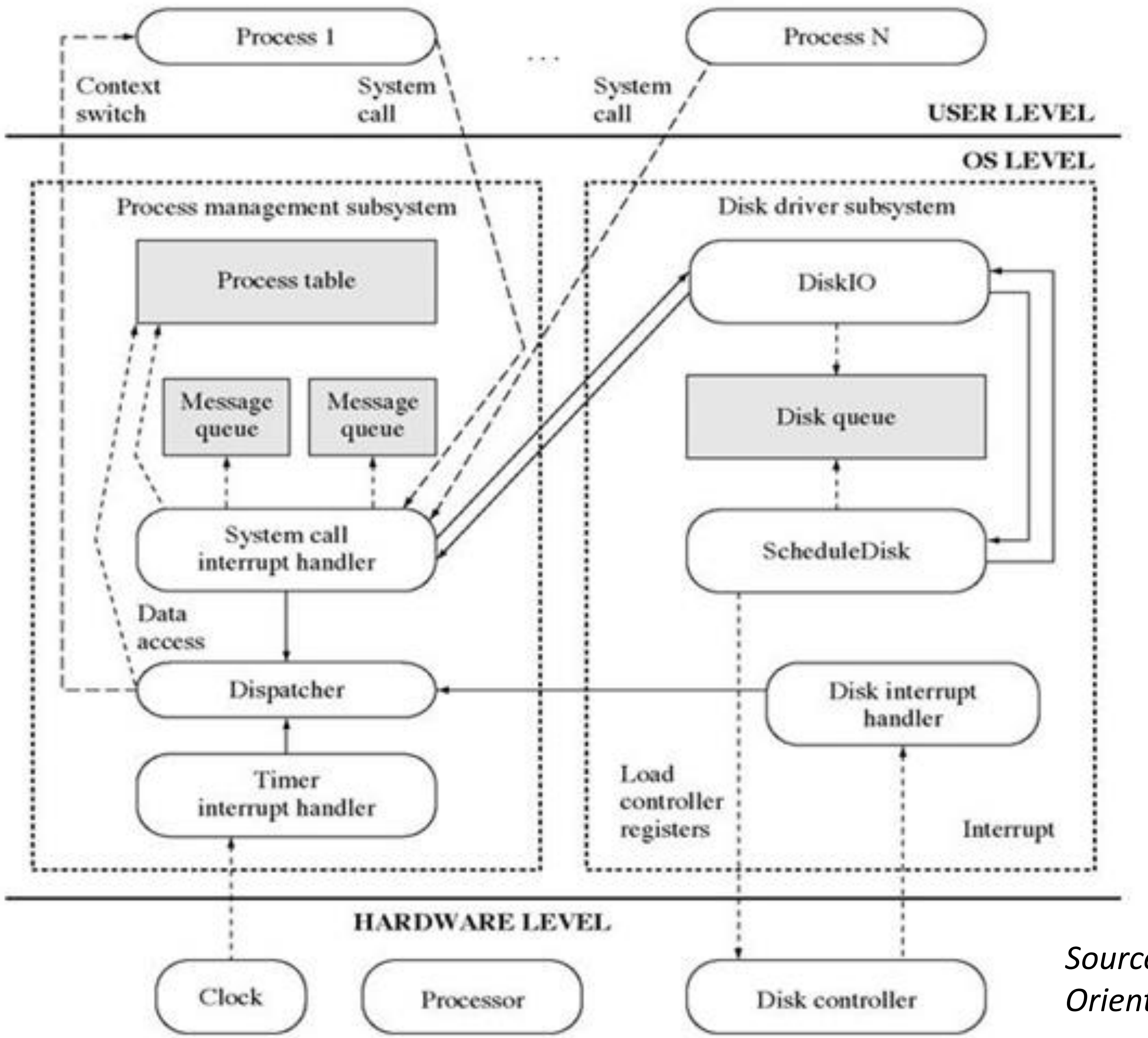


- When OS creates a NEW process
  - It creates a Process Control Block (PCB) – builds the data structures that are used to manage the process
  - Allocates address space in main memory

- PCB is "the manifestation of a process in an operating system”

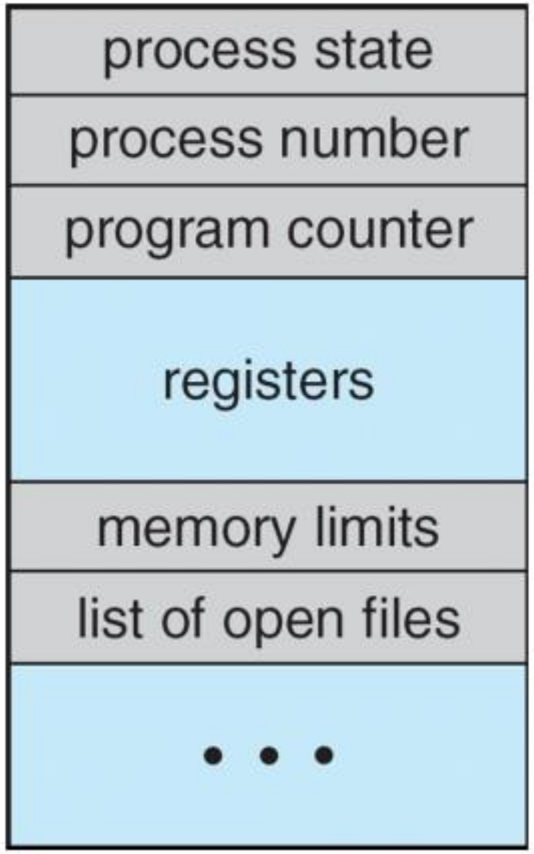


Source: http://www.technologyuk.net/computing/computer-software/operating-systems/



Source: Operating Systems: A Design-Oriented Approach by Charles Crowley

```
struct task_struct {
    volatile long state;    //The running state of the task (- 1 is not running, 0 is running (ready), > 0 has stopped).
    void *stack;            //Process Kernel Stack
    atomic_t usage;         //Several processes are using this structure
    unsigned int flags;     //per process flags, defined below// information about the state of the reaction process, but not the running state
    unsigned int ptrace;    //system call
    ......
    ......
};
```



Indicator: A unique identifier describing the process used to distinguish other processes.
Status: Task status, exit code, exit signal, etc.
Priority: Priority relative to other processes.
Program counter: The address of the next instruction to be executed in the program.
Memory pointers: pointers to program code and process-related data, as well as memory blocks shared with other processes
Context data: Data in the register of the processor when the process executes.
I/O status information: Includes displayed I/O requests, I/O devices assigned to processes, and a list of files used by processes.
Accounting information: It may include total processor time, total number of clocks used, time limit, account number, etc.

OS Control structures

OS maintains tables of information.

Memory Table

- Allocation of main and secondary memory for processes
- Shared memory regions
- Info about virtual memory

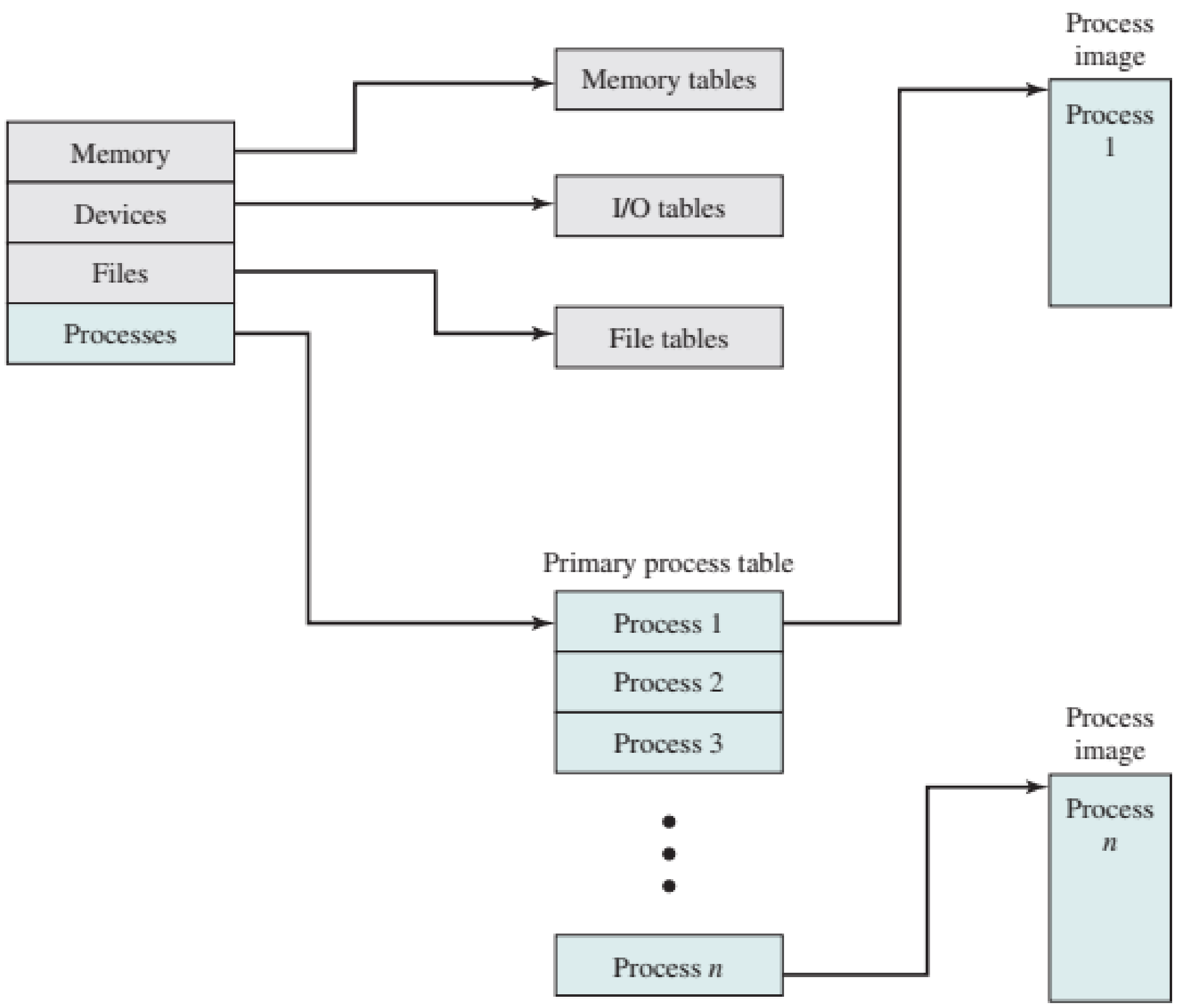
I/O Table

- Status of I/O devices and operations
- Location in main memory

File Table

- Location in secondary memory
- Existence of files
- Current status

All the tables are cross referenced



Process Control Block

- Process identification
  - An index into the process table
- Processor state information
  - Contents of processor registers
  - Processor by design has Program Status Word (PSW)
  - Reflects the current system state
- Process control information
  - For control and coordination of various processes

Process Identification

Identifiers

Numeric identifiers that may be stored with the process control block include

- Identifier of this process.
- Identifier of the process that created this process (parent process).
- User identifier.

Processor State Information

User-Visible Registers

A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.

Control and Status Registers

These are a variety of processor registers that are employed to control the operation of the processor. These include:

- **Program counter:** Contains the address of the next instruction to be fetched.
- **Condition codes:** Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow).
- **Status information:** Includes interrupt enabled/disabled flags, execution mode.

User visible registers

- Data registers and address registers (index register, segment pointer, and stack pointer)

Control and Status registers

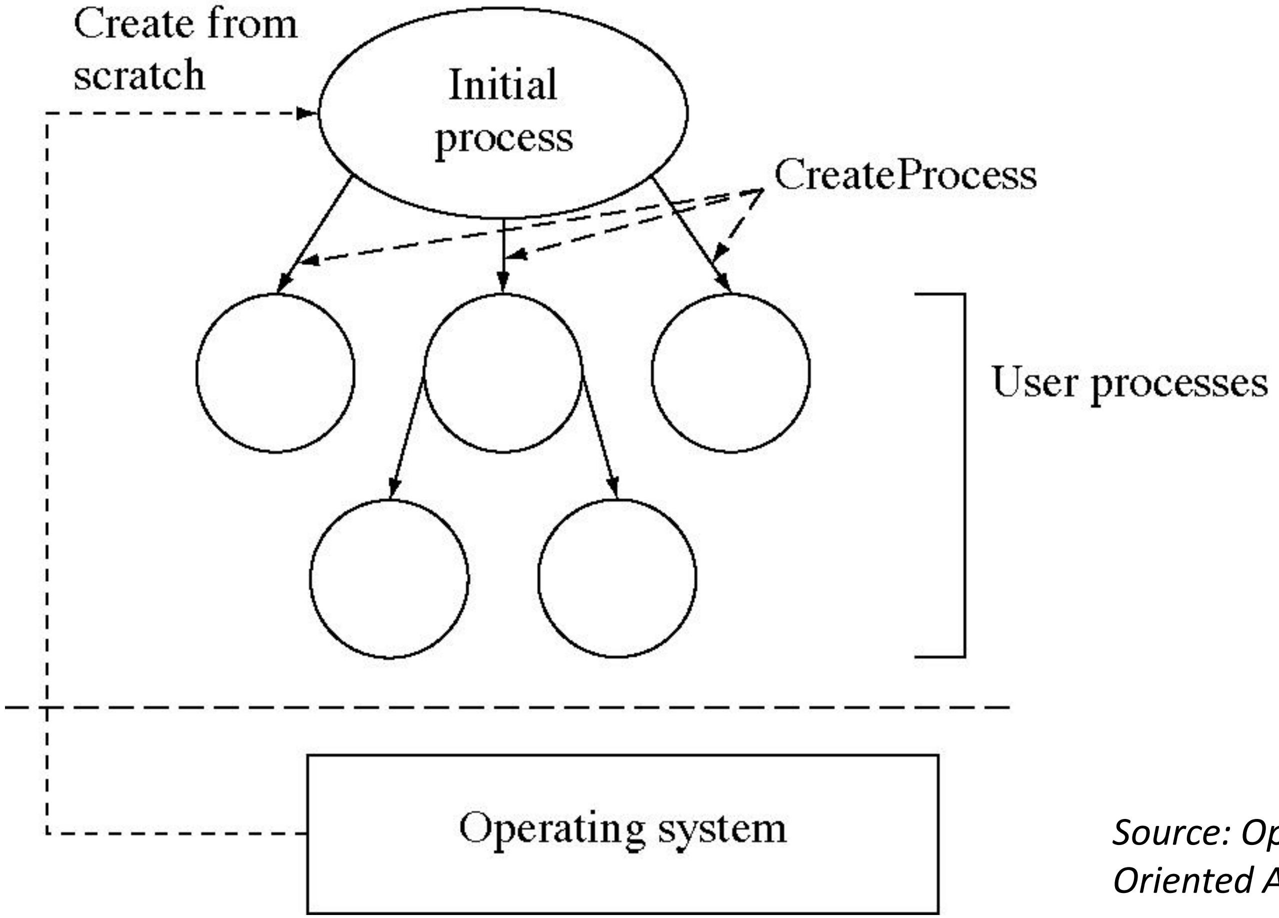
- Program counter (PC), Instruction Register (IR)

Process Control

- Modes of execution
  - User mode and Kernel mode
  - Kernel mode: complete control of processor, registers, instructions, and memory
  - Bit in the PSW indicates the mode of execution
- When interrupt occurs (or when an application program places a system call)
  - Processor clears the processor status register (that includes privilege level)
  - Sets privilege level to 0
  - Interrupt handling routine
  - IRET – restores the status register and the privilege level of the program

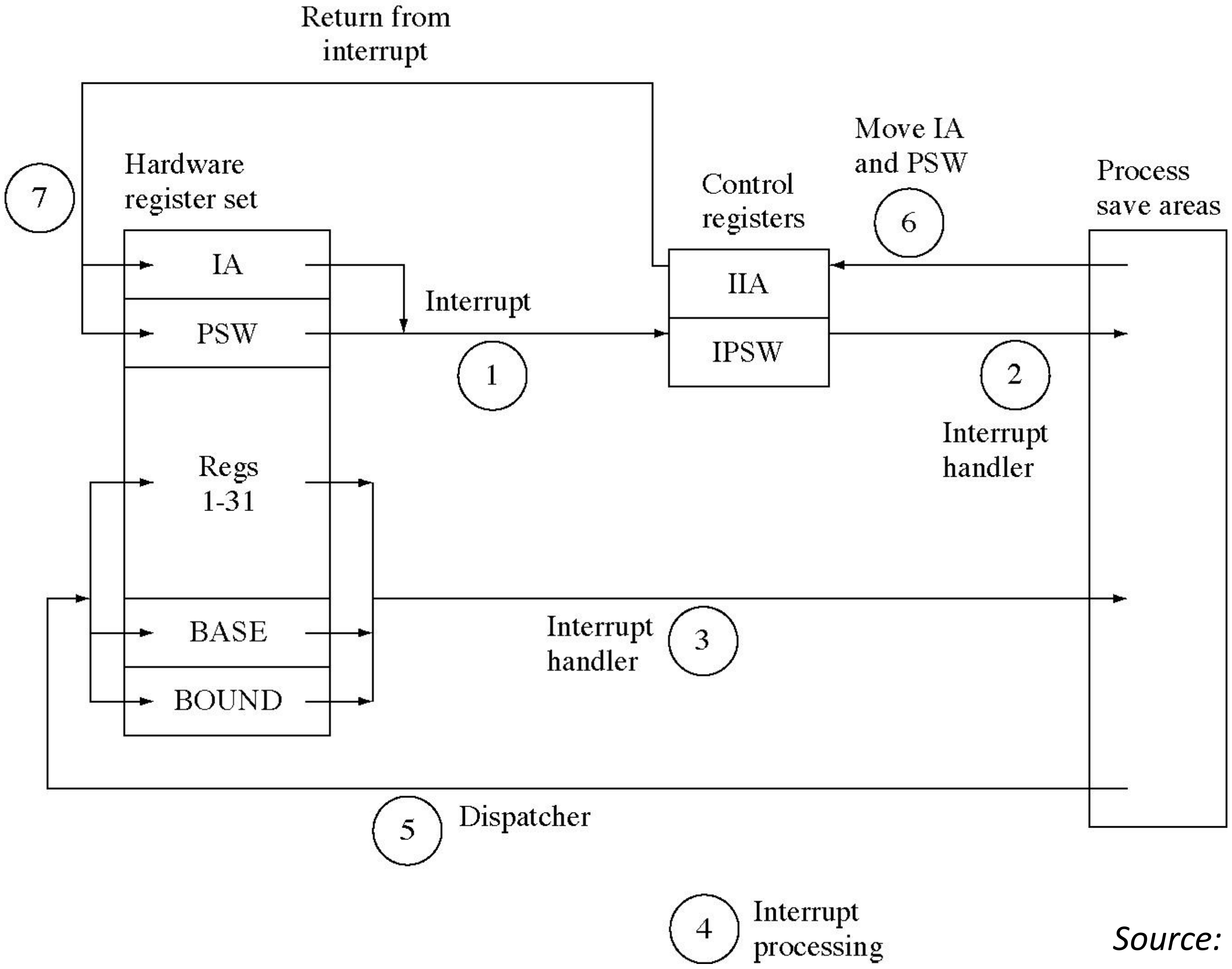
- Process Creation
  - Assign a unique process identifier to the new process
  - Allocate space for the process
  - Initialize the process control block - processor state information portion will typically be initialized with most entries zero, except for PC and SP
  - Set the appropriate linkages
  - Create or expand other data structures

Initial process creates other processes



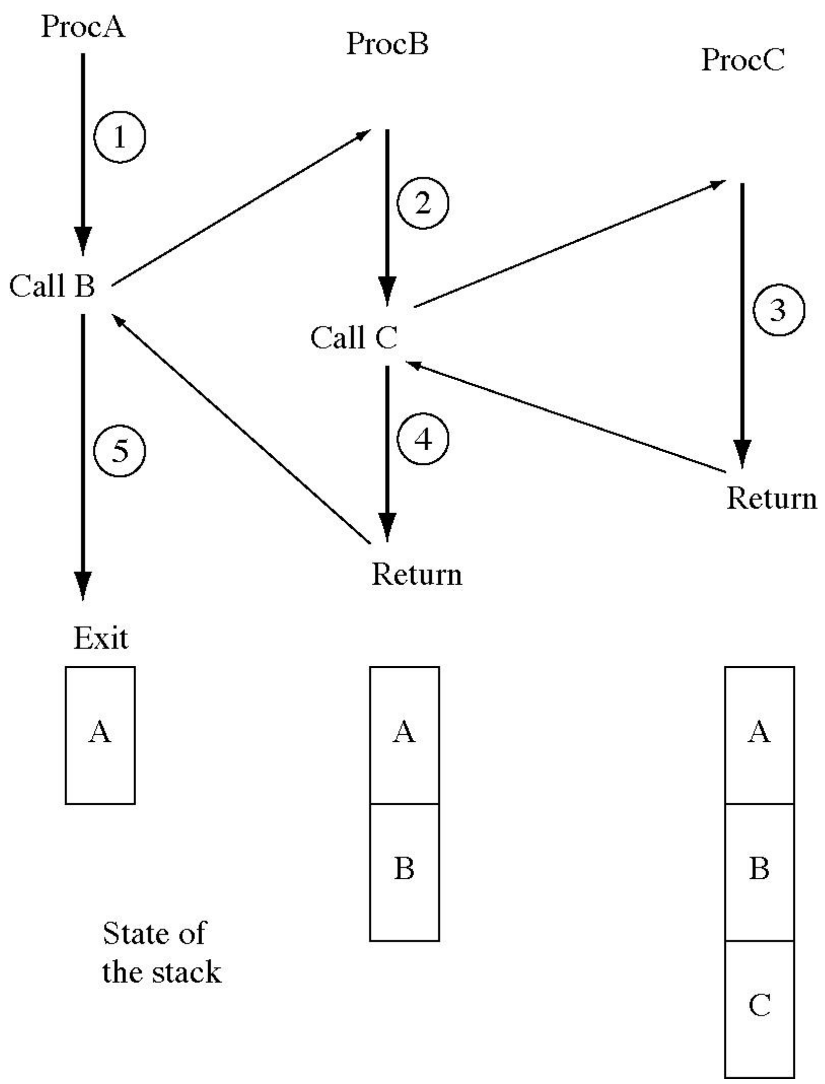
Source: Operating Systems: A Design-Oriented Approach by Charles Crowley

Process switching



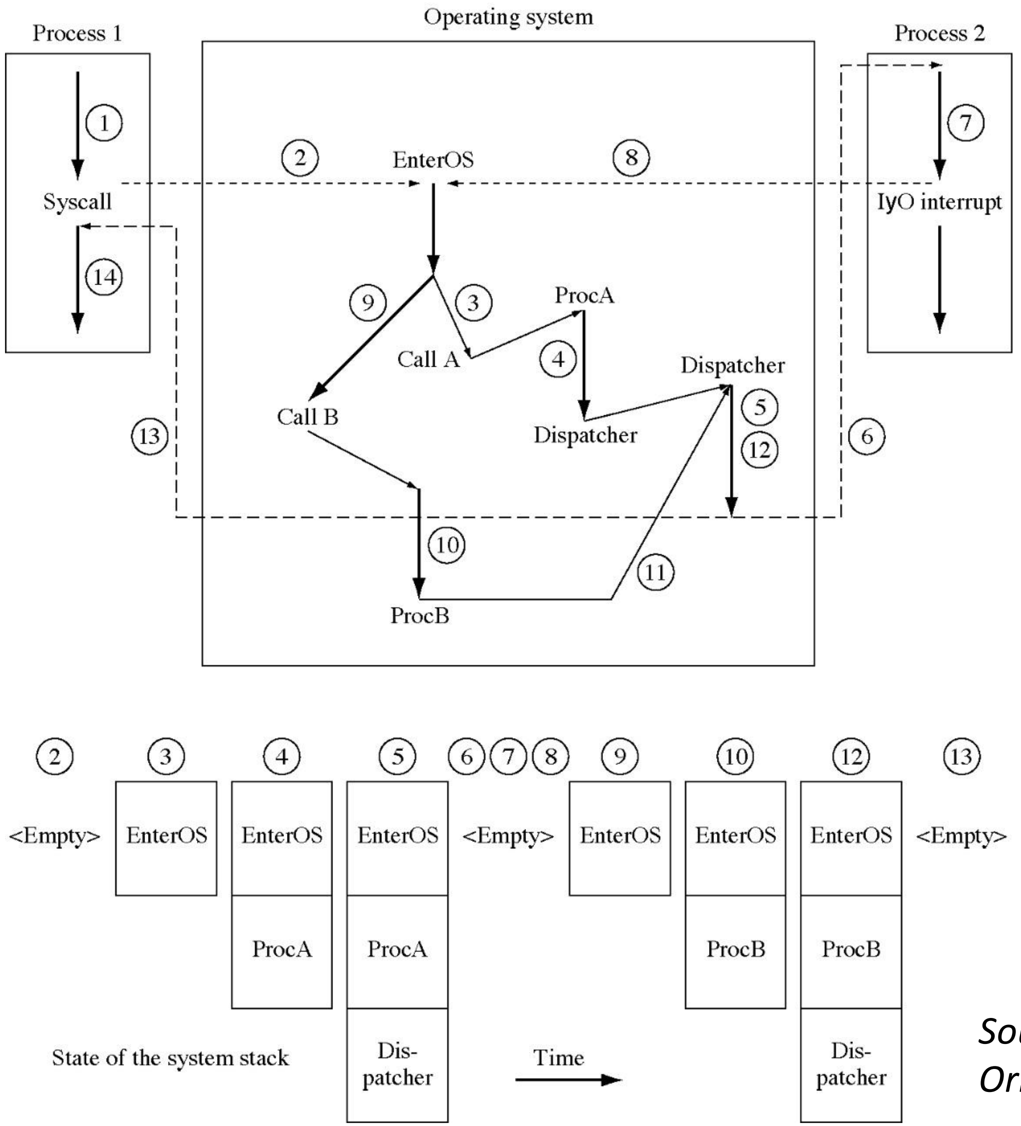
Source: Operating Systems: A Design-Oriented Approach by Charles Crowley

Flow of control within a process



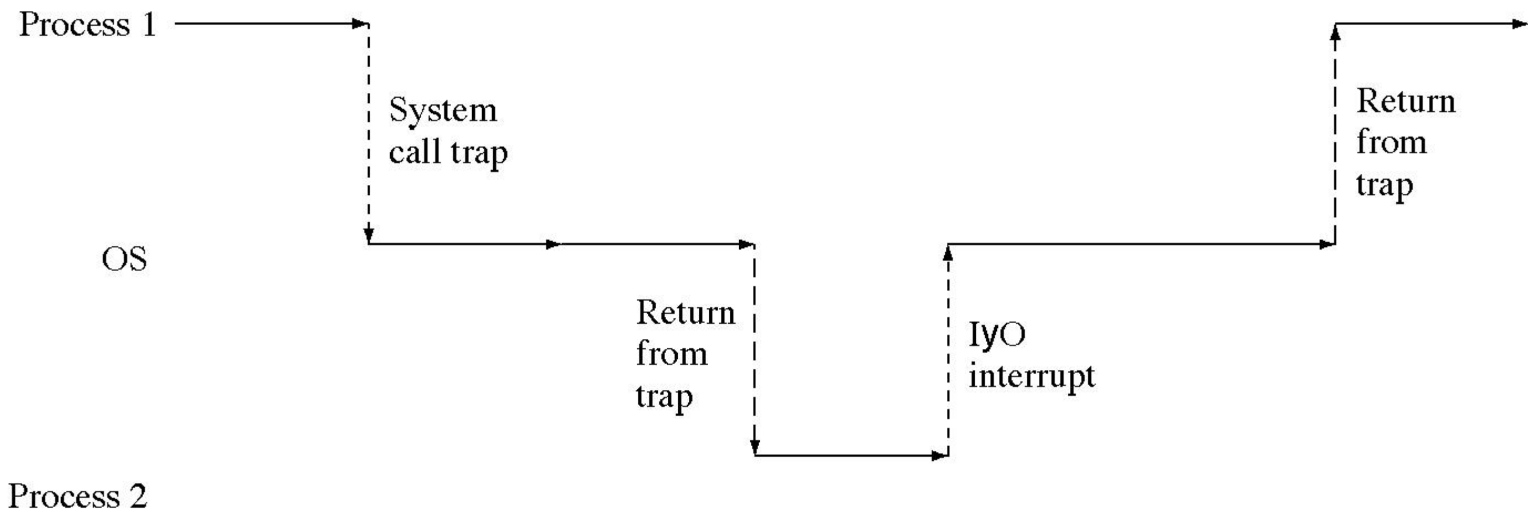
Source: Operating Systems: A Design-Oriented Approach by Charles Crowley

Process switching control flow



Source: Operating Systems: A Design-Oriented Approach by Charles Crowley

Flow of control during process switching (another view)



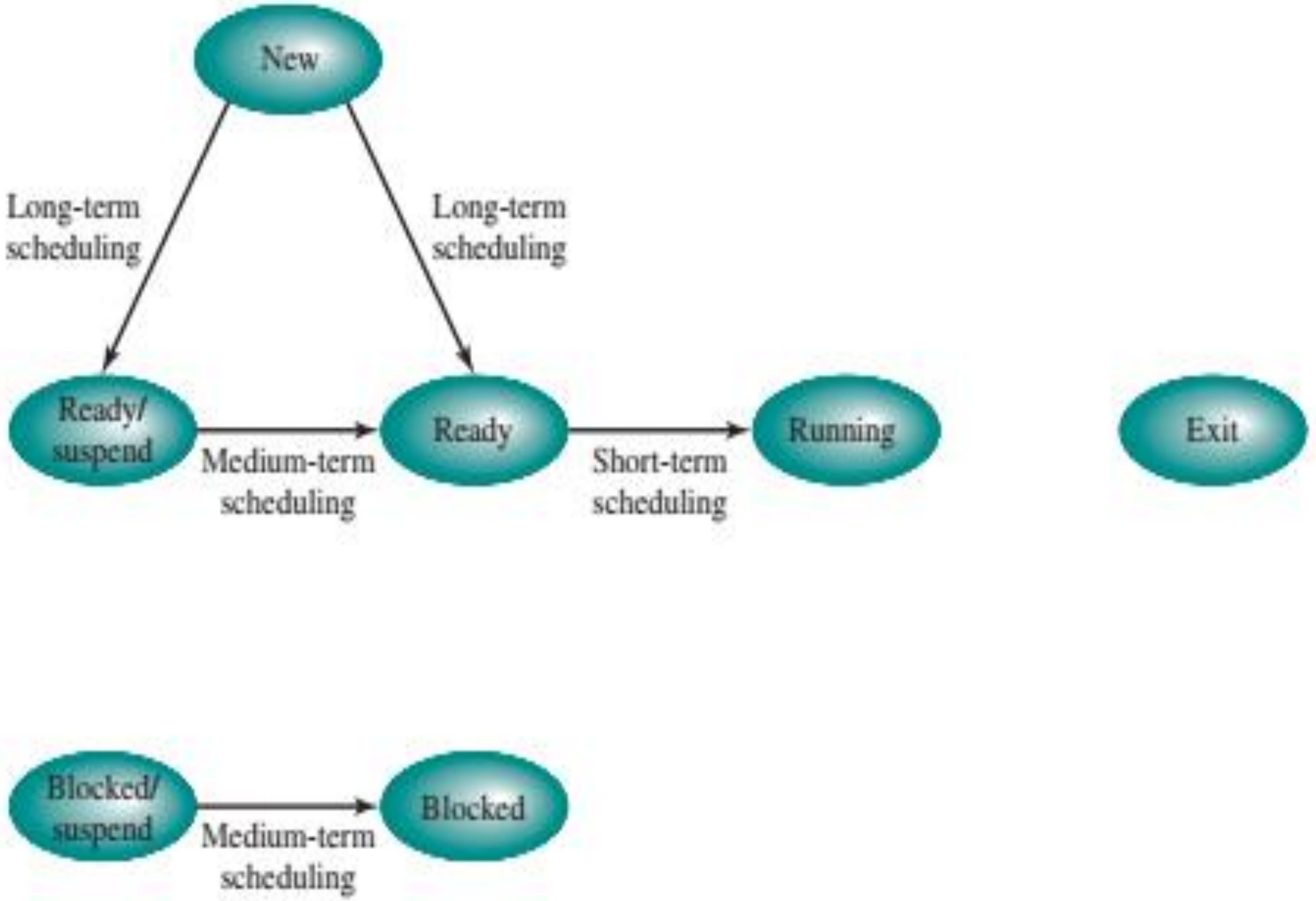
Source: Operating Systems: A Design-Oriented Approach by Charles Crowley

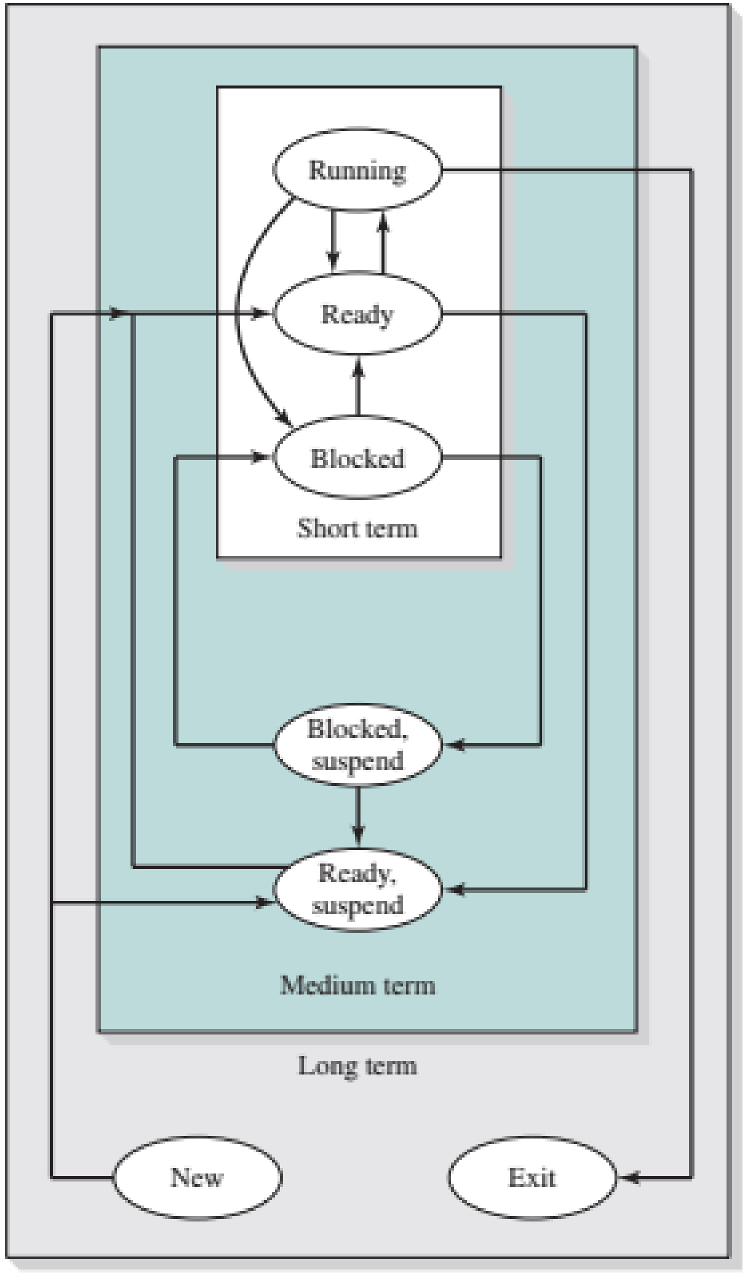
Scheduling

- Key to Multi programming
- Multiple processes exist concurrently in main memory
- Achieve system objectives such as response time, throughput, processor efficiency
- Types of scheduling
  - Long-term scheduling
  - Medium-tem scheduling
  - Short-term scheduling

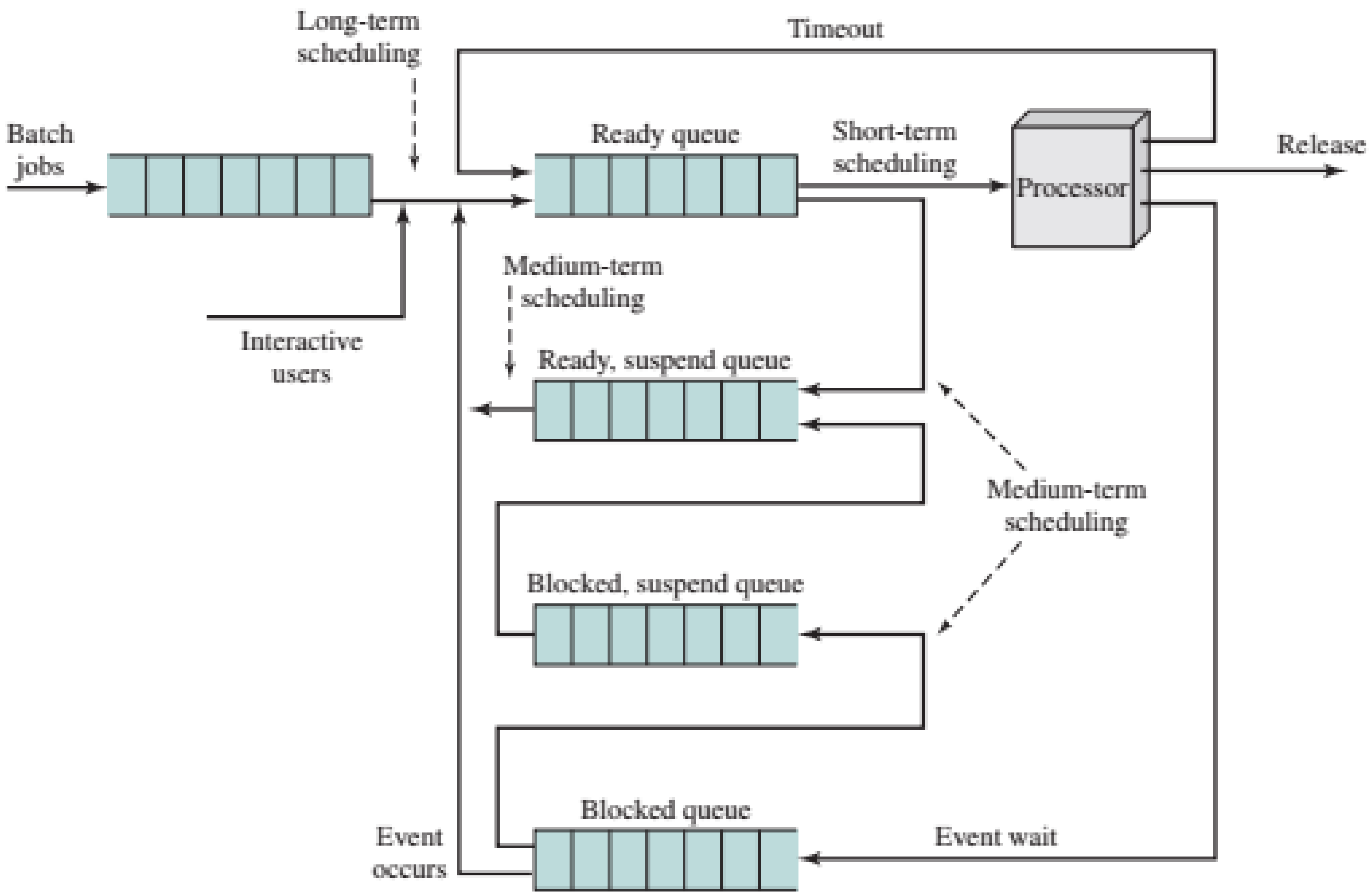
Types of Scheduling

- Long term scheduling
  - decision to add to the pool of processes to be executed
- Medium term scheduling
  - decision to add to the number of processes that are partially or fully in main memory
- Short term scheduling
  - decision as to which available process will be executed by the processor


New
Long-term
scheduling
Long-term
scheduling
Ready/
suspend
Medium-term
scheduling
Ready
Short-term
scheduling
Running
Exit
Blocked/
suspend
Medium-term
scheduling
Blocked


Running
Ready
Blocked
Short term
Blocked,
suspend
Ready,
suspend
Medium term
Long term
New
Exit

Queueing diagram



Long term scheduling

- Determines which process is admitted to the system for processing
- Decision making
  - When the OS to add one or more processes
  - Which jobs to turn into processes
- More process creation, small percentage of CPU time devoted for each process
- Goal is to provide satisfactory service to current set of processes
  - Process termination, news jobs are added
  - Processor is idle, long term scheduler is invoked
- FCFS – First Come, First Serve basis
  - Criteria can be priority, expected execution time, I/O requirements
  - Processor bound and I/O bound processes

Medium term scheduling

- Swapping in and out of processes
- In tandem with memory management system
  - Virtual memory

Short term scheduling

- Long-term: infrequent decision, more coarse grained decision whether to admit a process or not.
- Medium-term: more frequently than long-term; swapping in and out

- Dispatcher
  - Executes most frequently; most fine grained decision of which process to be executed next
  - Interrupts and Traps
    - Clock interrupts, I/O interrupts, OS calls, etc.

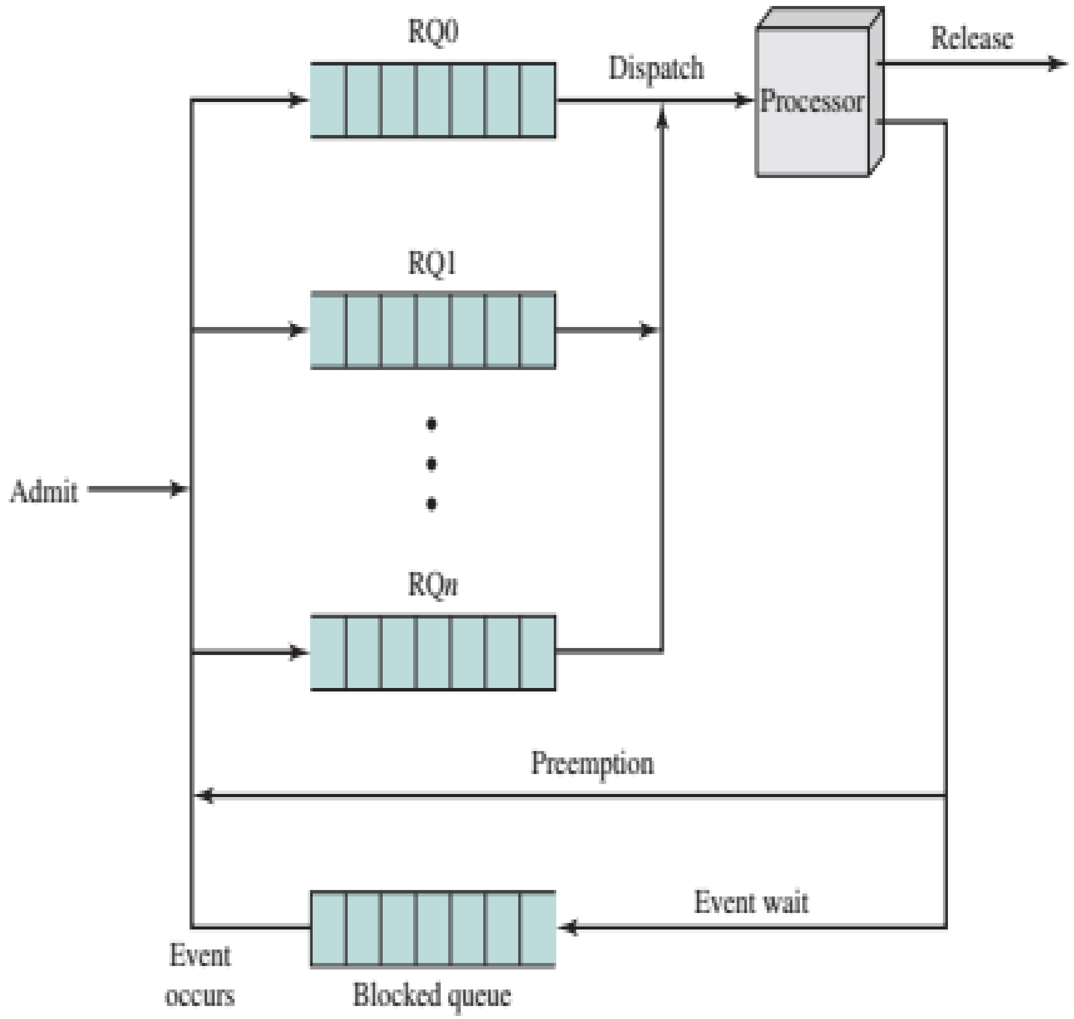
Scheduling algorithms

- Criteria
  - User-oriented criteria
    - Response time: time from the submission of a request until the response begins to be received
    - Turnaround time: interval of time between the submission of a process and its completion
  - System-orientedcriteria
    - Throughput: maximize the number of processes completed per unit of time
    - Processor utilization: percentage of time that the processor is busy
    - Fairness: processes should be treated the same, and no process should suffer starvation
    - Enforcing priorities
    - Balancing resources: resources of the system should be kept busy

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Priority Queueing

- Low priority processes may suffer starvation
- Priority of process can change with its age or execution history



- Selection function determines which among the processes should be selected as the next process for execution
  - Based on priority, resource requirement, etc.
- Quantifiers
  - *w* = time spent in system so far, waiting and executing
  - *e* = time spent in execution so far
  - *s* = total service time required by the process, including *e*
- Decision mode
  - Non-preemptive: once a process is in the Running state, it continues to execute until (a) it terminates or (b) it blocks itself to wait for I/O or to request some OS service.
  - Pre-emptive: running process may be interrupted and moved to the Ready state by the OS
    - Decision to preempt: a new high priority process arrives
    - an interrupt occurs that places a blocked process in the Ready state
    - periodically, based on a clock interrupt

- Preemptive policies incur greater overhead than non-preemptive ones, but may provide better service to the total population of processes
  - prevent any one process from monopolizing the processor for very long
- Cost is low
  - using efficient process-switching mechanisms (as much help from hardware as possible) and by providing a large main memory to keep a high percentage of programs in main memory

	FCFS	Round Robin	SPN	SRT	HRRN	Feedback
Selection Function	$\max[w]$	constant	$\min[s]$	$\min[s - e]$	$\max\left(\frac{w + s}{s}\right)$	(see text)
Decision Mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Throughput	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response Time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on Processes	Penalizes short processes; penalizes I/O-bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O-bound processes
Starvation	No	No	Possible	Possible	No	Possible

SPN: Shortest Process Next
SRT: Shortest Remaining Time
HRRN: Highest Response Ratio Next

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

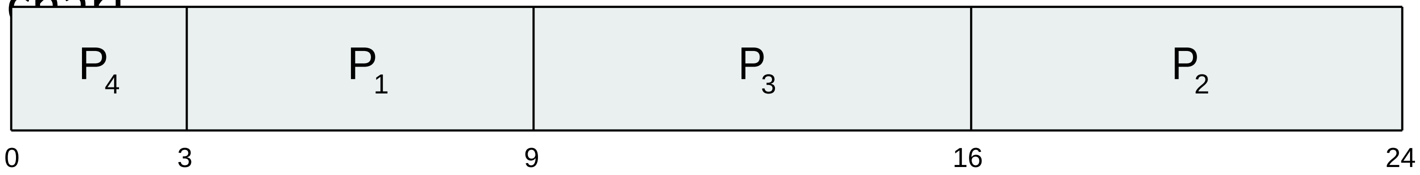
- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6;$ $P_2 = 0_;$ $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Multi Level Feedback Queue (MLFQ) Scheduling

- optimize turnaround time
- minimize response time - system feel responsive to interactive users

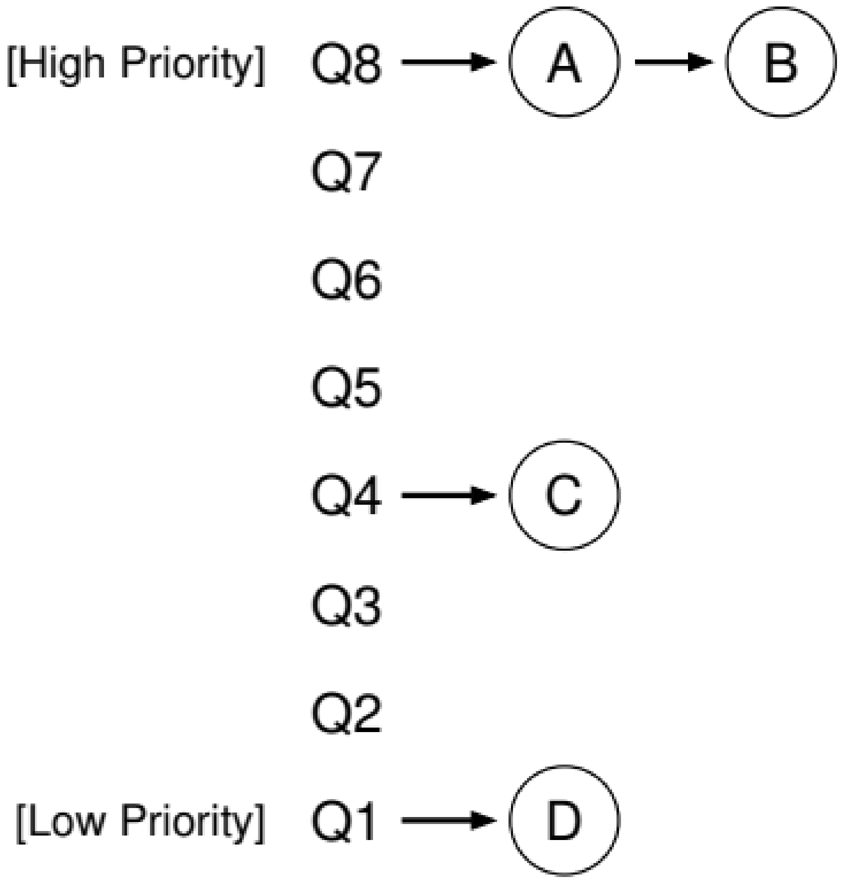
- number of distinct queues each assigned a different priority level
- a job that is ready to run is on a single queue

first two basic rules for MLFQ:

Rule 1: If Priority(A) > Priority(B), A runs (B doesn't)

Rule 2: If Priority(A) = Priority(B), A & B run in RR.

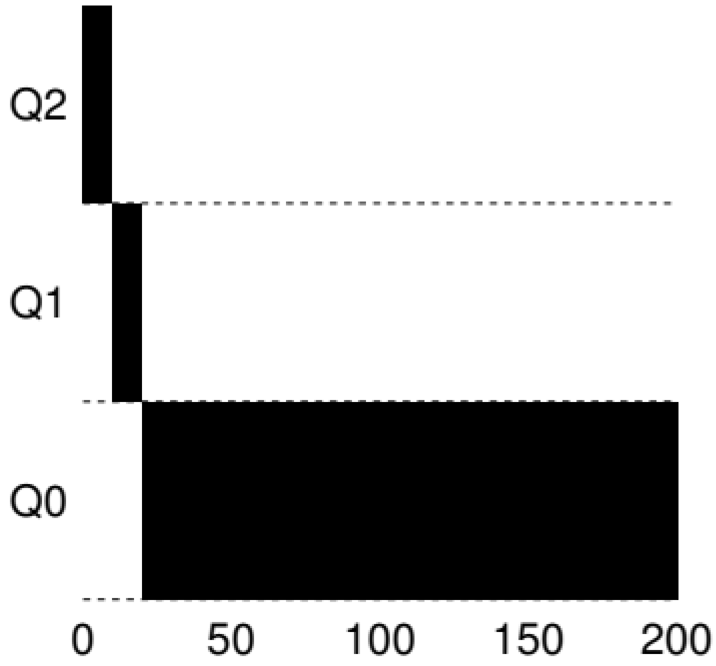
- MLFQ varies the priority of a job based on its observed behavior
  - If a job repeatedly relinquishes the CPU while waiting for input from the keyboard, **MLFQ will keep its priority high**, as this is how an interactive process might behave
  - If, instead, a job uses the CPU intensively for long periods of time, **MLFQ will reduce its priority**



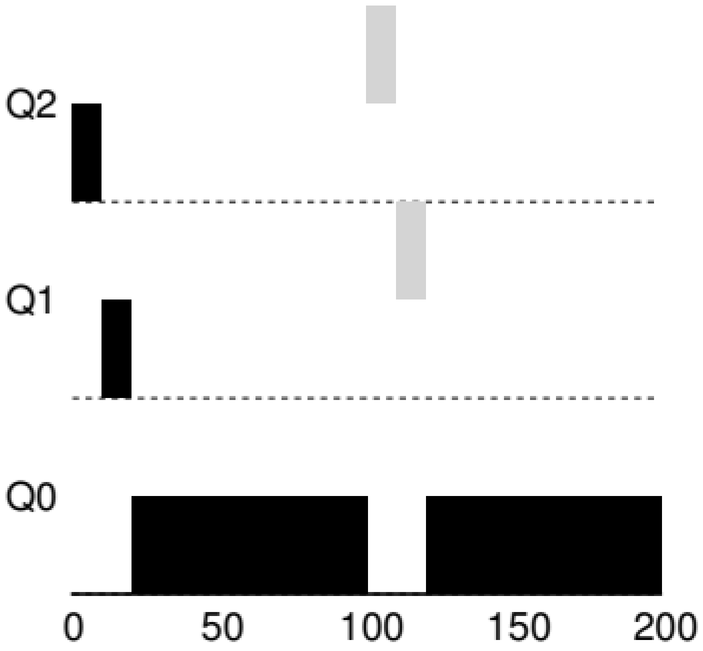
Changing Priority

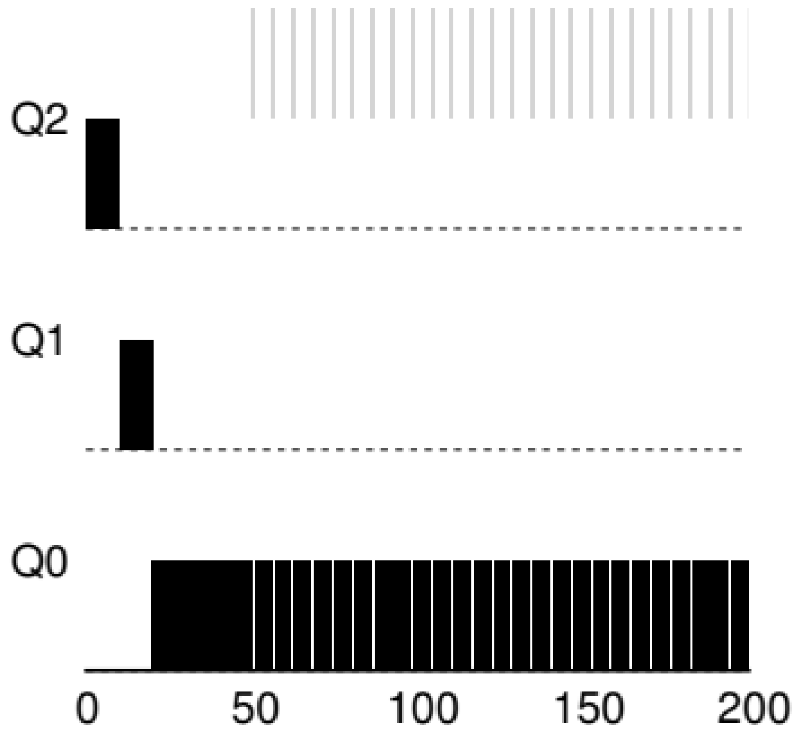
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue)
- **Rule 4a:** If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue).
- **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the *same* priority level.

Long Process A (CPU intensive)



Short running interactive Process B

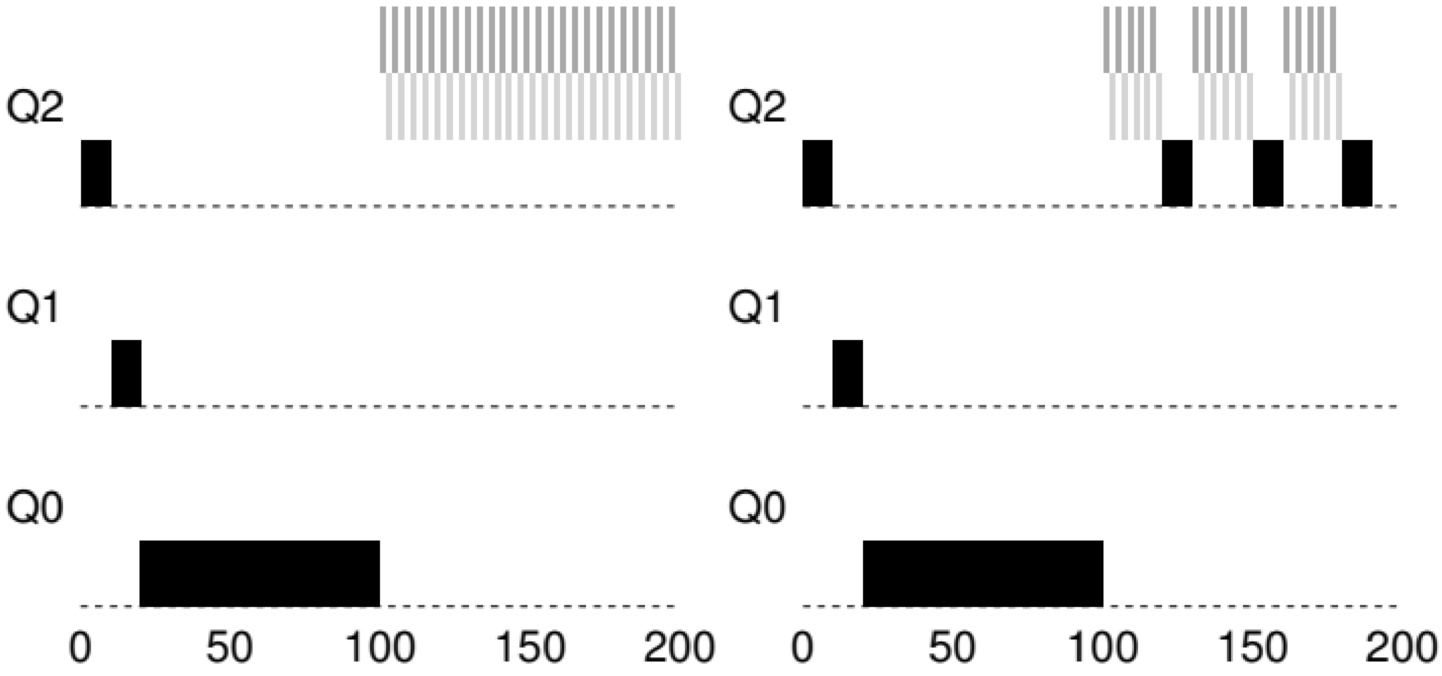




- Mixture of I/O-intensive and CPU-intensive Workload
- Interactive Process B (gray) that uses CPU for only 1 ms

Priority Boost

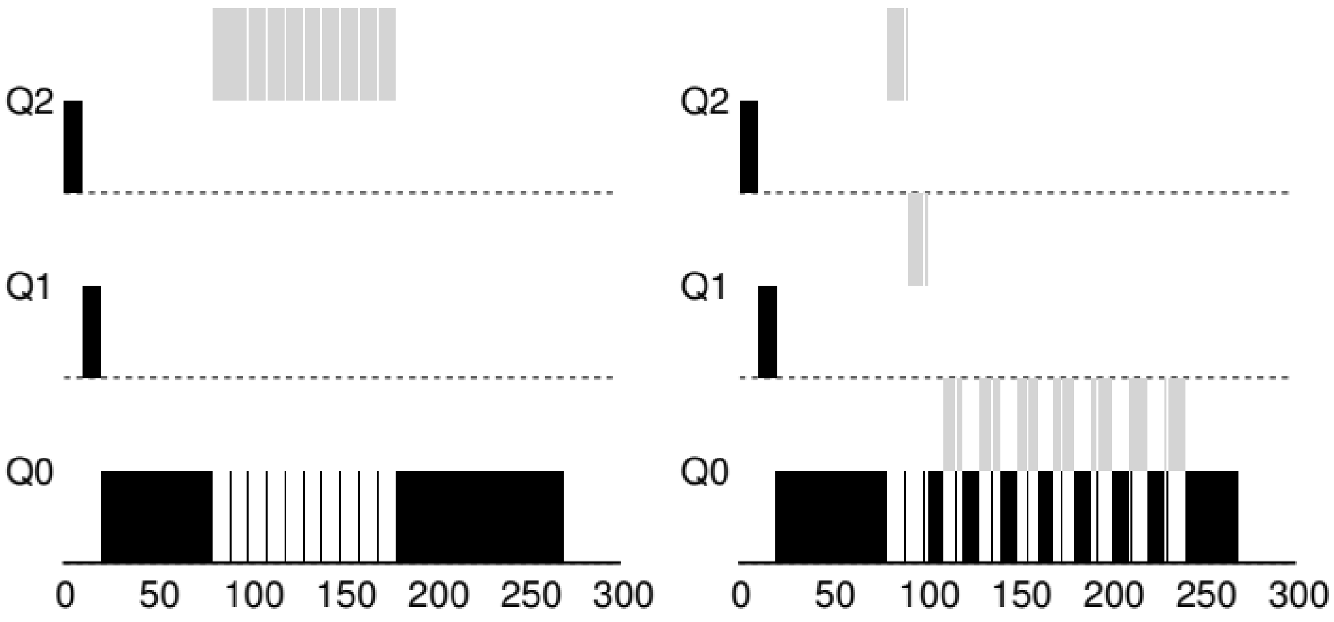
- Move all the jobs to the topmost queue after time period.
- Advantages:
  - processes are guaranteed not to starve



- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

What is the value for S?

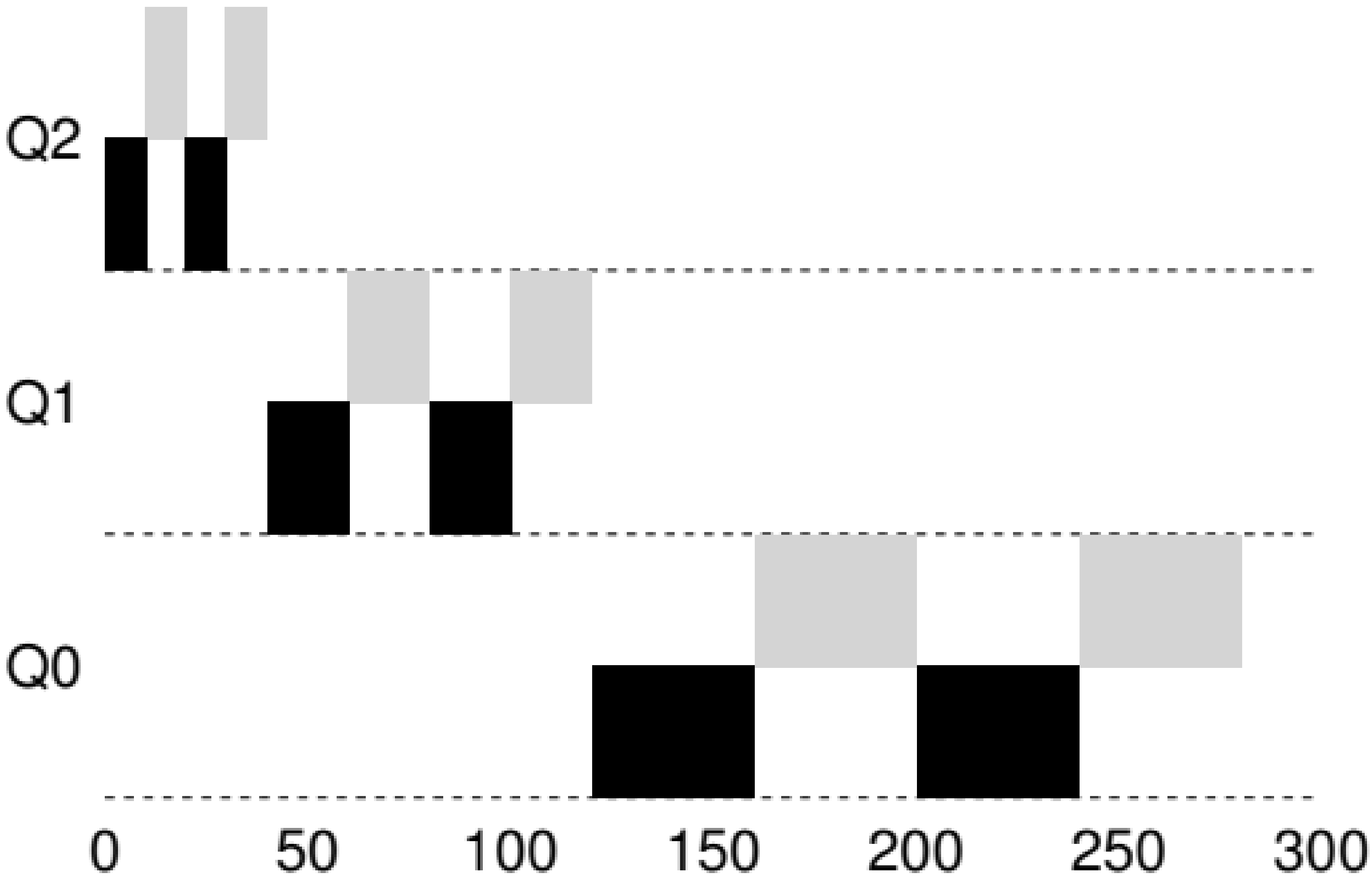
Gaming the scheduler



- generally refers to the idea of doing something sneaky to trick the scheduler into giving you more than your fair share of the resource
- before the time slice is over, issue an I/O operation (to some file you don't care about) and thus relinquish the CPU; doing so allows you to remain in the same queue, and thus gain a higher percentage of CPU time
- thereby, a job could nearly monopolize the CPU

- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue)

Lower Priority, Longer Quanta



Summary - MLFQ

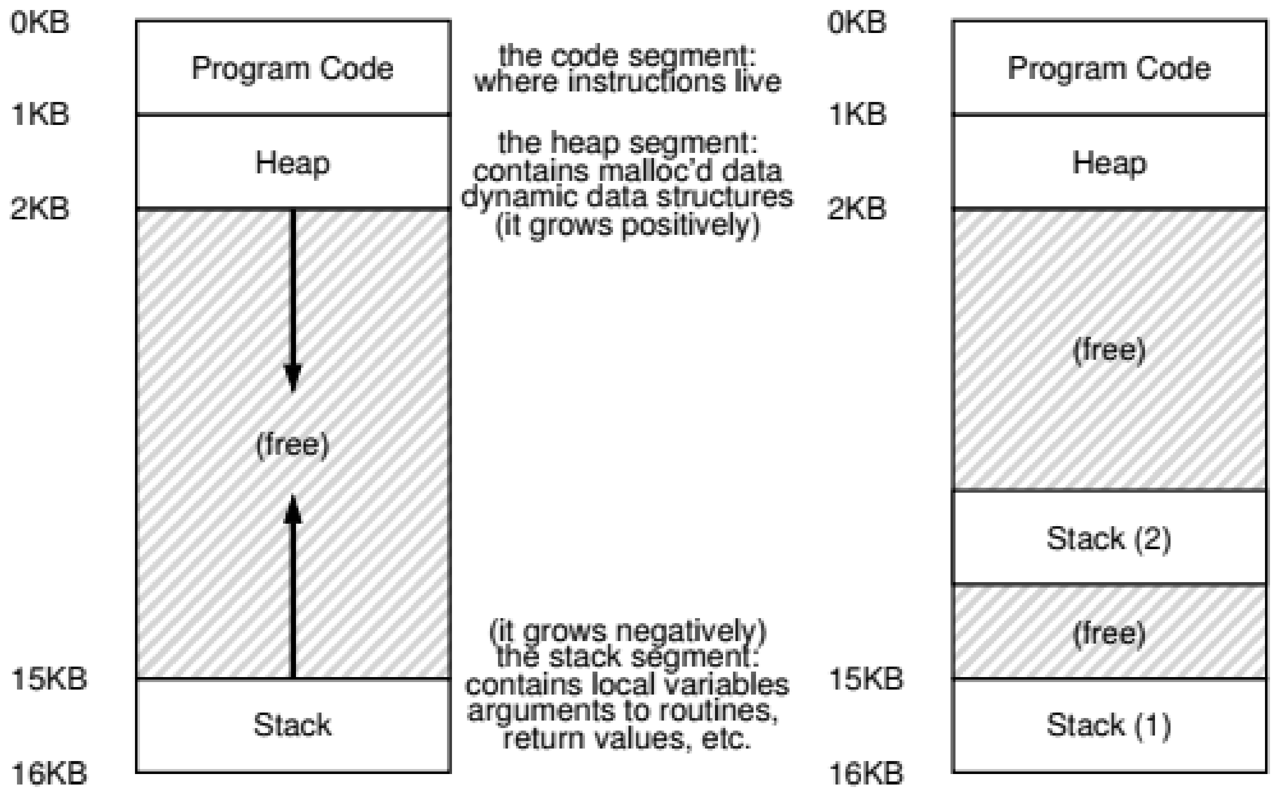
- **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
- **Rule 2:** If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.

Threads

- Concept of Process
  - **Resource ownership**: A process includes a virtual address space to hold the process image, i.e., the collection of program, data, stack, and attributes defined in the PCB. Resources are main memory, Disk I/O, I/O devices, and files
  - **Scheduling/execution**: The execution of a process follows an execution path (trace) through one or more programs. A process has an execution state (Running, Ready, etc.) and a dispatching priority
- Threading
  - Ability of an OS to support multiple, concurrent paths of execution within a single process.
  - Lightweight process
  - Achieves parallelism

Threads … Contd.

- Single point of execution within a program
- Share same address space and can access same data
- Context switching: between threads; Adv.: address space remains the same, page table too
- Supports parallelism



Single-Threaded and Multi-Threaded Address Spaces

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

- Main program creates two threads
- pthread_create() : Creates thread
- Pthread_join(): waits for a particular thread to complete

Thread trace (1)

main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
		runs
		prints “B”
		returns
prints “main: end”		

Thread trace (2)

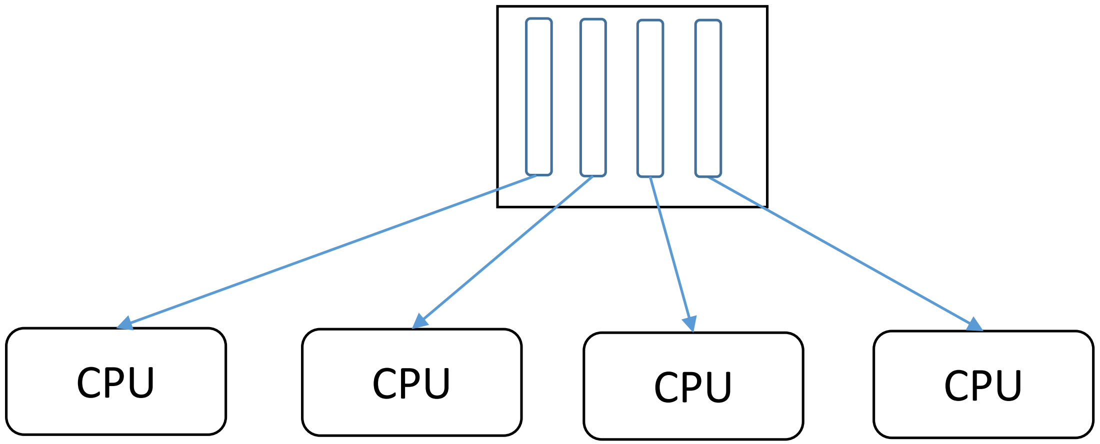
main	Thread 1	Thread2
starts running		
prints “main: begin”		
creates Thread 1		
	runs	
	prints “A”	
	returns	
creates Thread 2		
		runs
		prints “B”
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints “main: end”		

Thread trace (3)

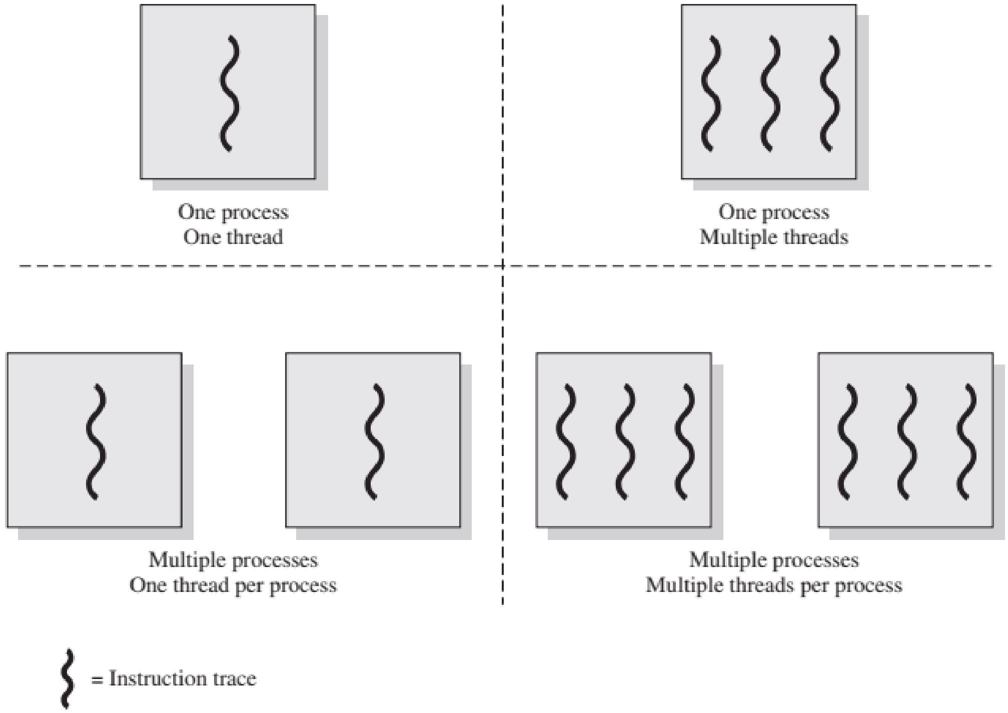
main	Thread 1	Thread2
starts running		
prints "main: begin"		
creates Thread 1		
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
	runs	
	prints "A"	
	returns	
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

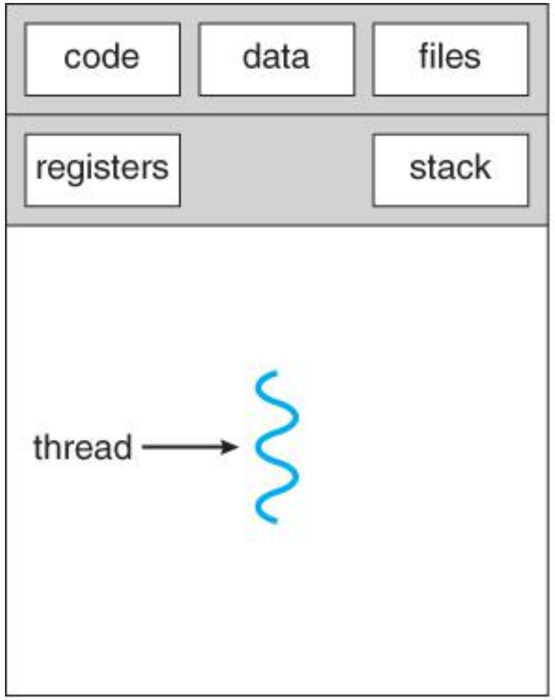
Threads

- Four threads created
- Each thread is independent
- Management of threads is simpler than processes
- Shared instructions, global, and heap regions
- Each thread has its own stack

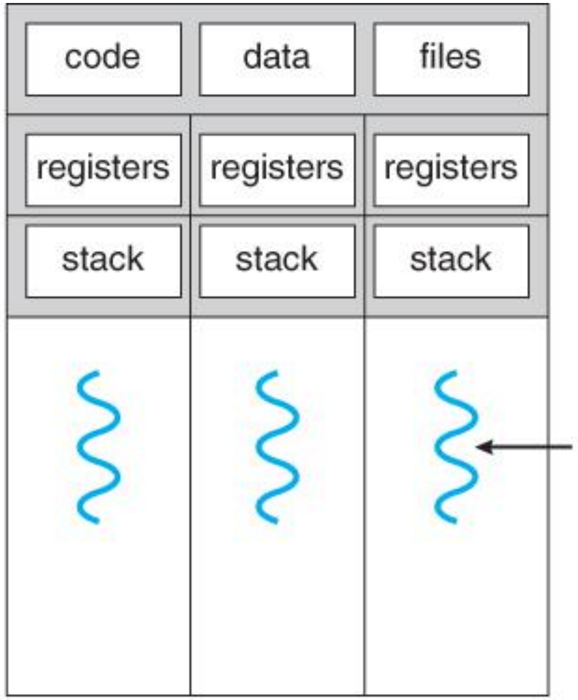


- Process:
  - A virtual address space that holds the process image
  - Protected access to processors, other processes, files, and I/O resources
- Threads
  - A thread execution state (Running, Ready, etc.)
  - A saved thread context when not running; one way to view a thread is as a independent program counter operating within a process
  - An execution stack
  - Some per-thread static storage for local variables
  - Access to the memory and resources of its process, shared with all other threads in that process





single-threaded process



multithreaded process

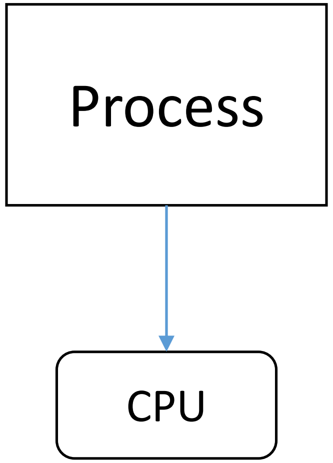
Per process	Per thread
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Signals and signal handlers	
Accounting info	

POSIX threads – IEEE 1003.1c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NTHREADS 10
void *print_hello_world(void *tid)
{
        /* This function prints the thread’s identifier and then exits. */
        printf("Hello World. Greetings from thread %d\n", tid);
        pthread_exit(NULL);
}
```

```
int main (int argc, char *argv[]) {
    /* The main program creates 10 threads and then exits. */
    pthread_t threads[NTHREADS];
    int status, i;
    for(i=0; i < NTHREADS; i++) {
        printf("Main here. Creating thread %d\n", i);
    status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
    if (status != 0) {
        printf("pthread returned error code %d\n", status);
        exit(-1);
    }
    }
    exit(NULL);
}
```

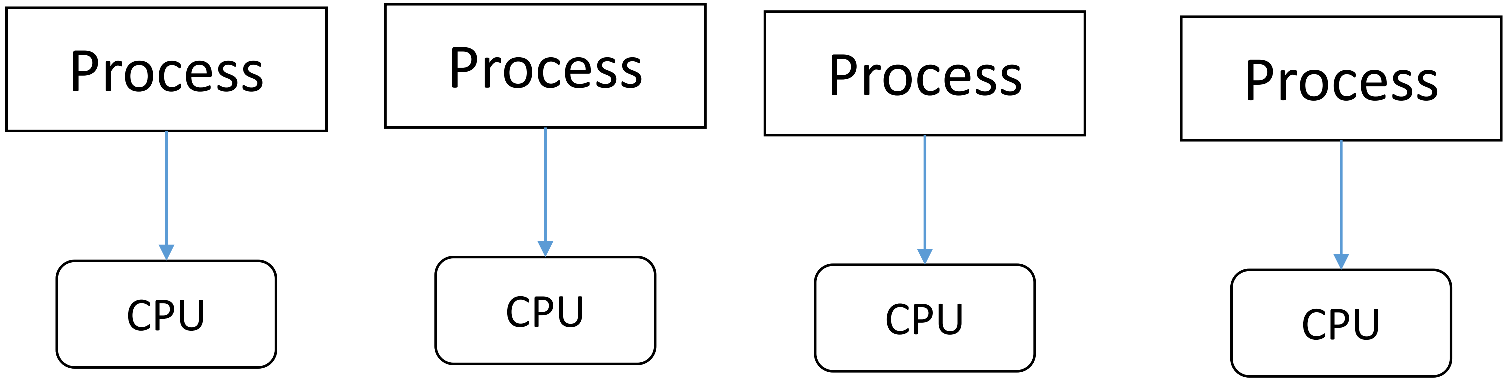
Sum of first 1,00,00,000 numbers



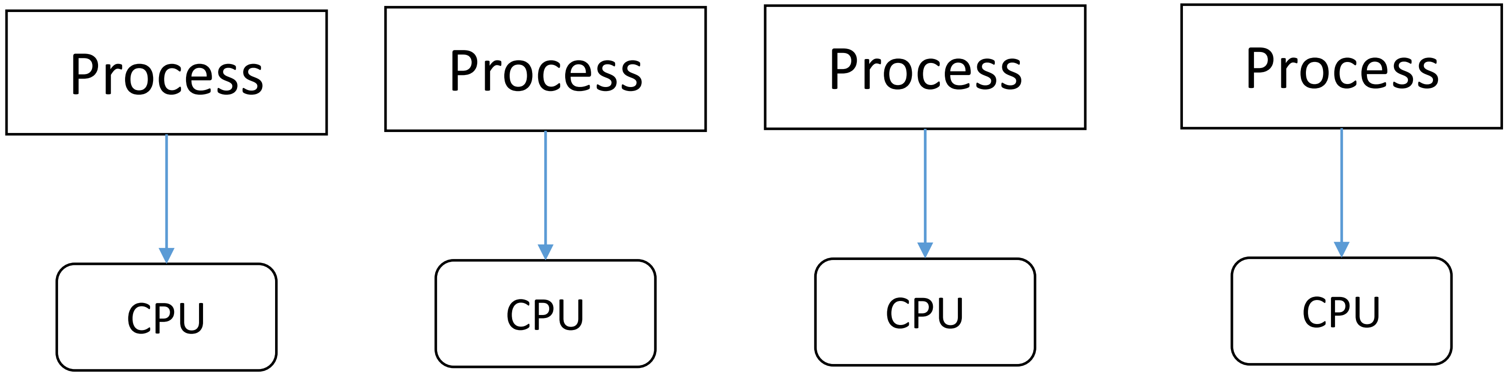

```
#include<stdio.h>
long add() {
        int i=0;
        long sum=0;

        while(i < 10000000) {
         sum = sum+=i;
         i++
        }
        return sum;
}
```

```
int main() {
        long sum;
        sum = add();
        printf("%l", sum);
}
```

Is it possible to speed up the operation? How?
Assume multiple processors/CPU

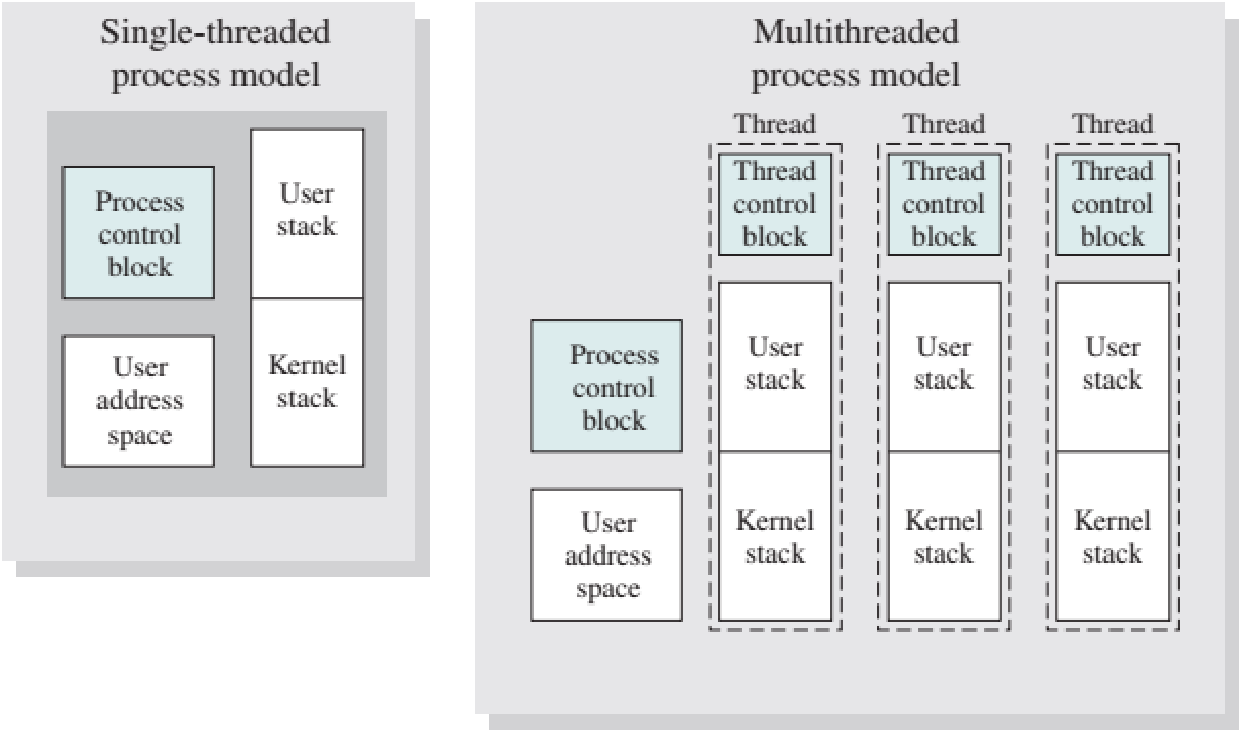


- Four fork() system calls; one for each process
- Each process executes independently
- IPC mechanism to communicate between processes
- Each process has its own instruction, data, heap, and stack



Benefits of Threads

- Far less time to create a new thread in an existing process, than to create a brand-new process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process than to switch between processes
- Threads enhance efficiency in communication between different executing programs



Threads vs Processes

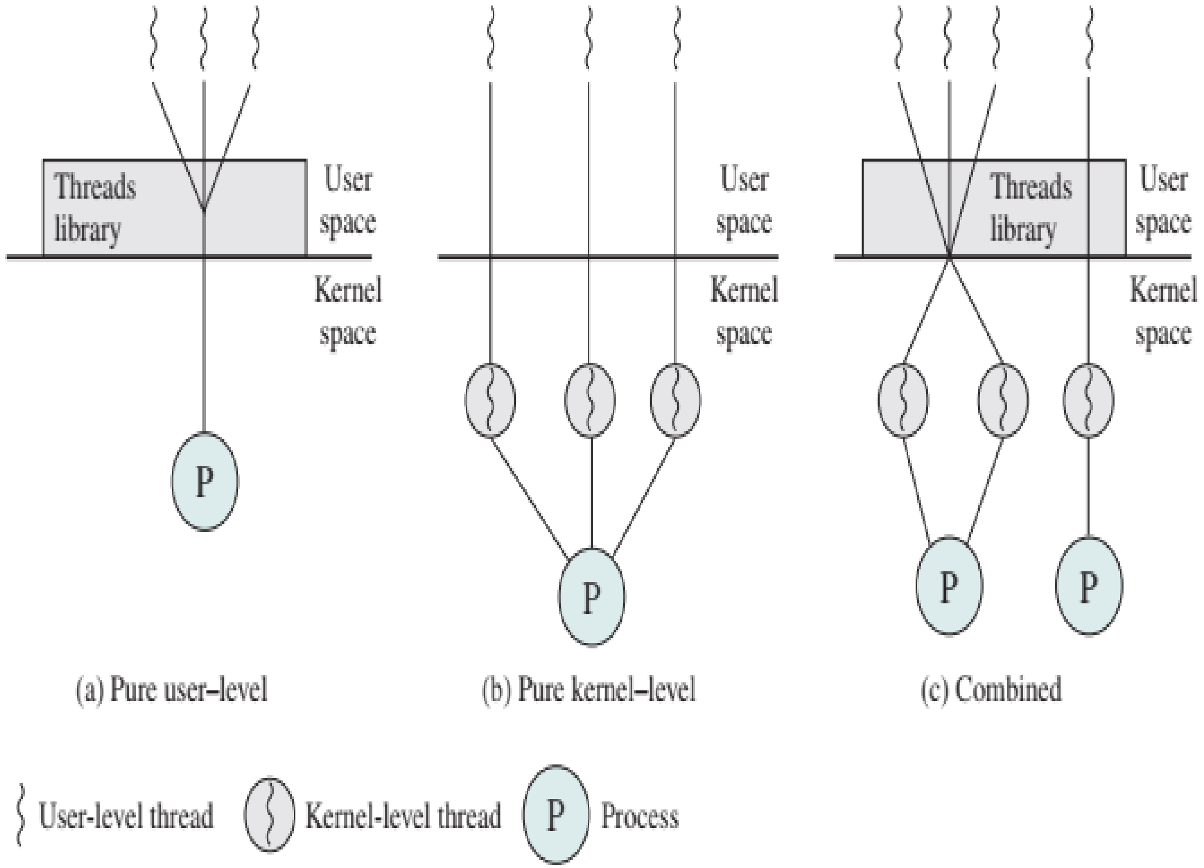
- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- There can be more than one thread in a process, the first thread calls main() & has the process's stack
- Inexpensive creation
- Inexpensive context switching
- Efficient communication
- If a thread dies, its stack is reclaimed

- A process has code/data/heap & other segments
- A process has at least one thread
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- Expensive creation
- Expensive context switching
- Interprocess communication can be expressive
- If a process dies, its resources are reclaimed & all threads die

Source: http://www.cs.columbia.edu/~junfeng/13fa-w4118/lectures/l08-thread.pdf

Types of Threads

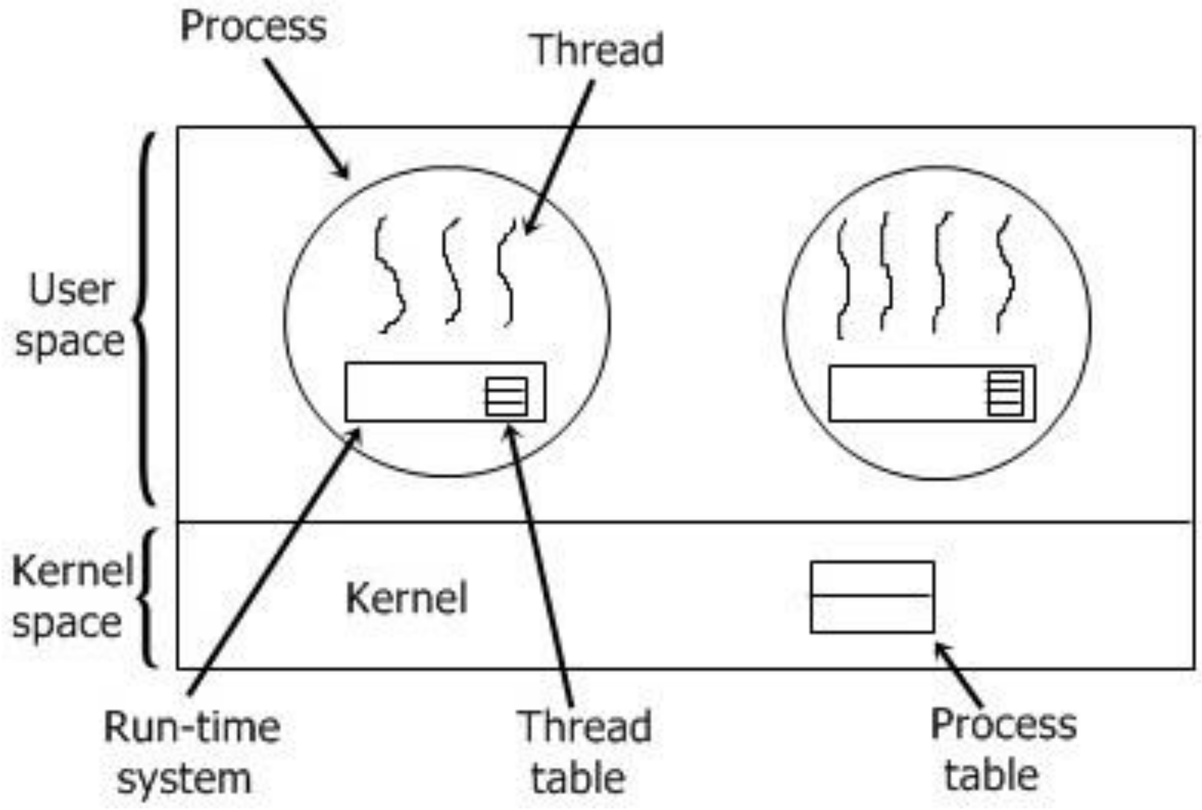
- User-Level Threads
  - thread management is done by a user level thread library
  - the kernel does not know anything about the threads running
- Kernel-Level Threads
  - threads are directly supported by the kernels
  - also known as light weight processes



User Level Threads

- Fast as no system call to manage. Thread library does everything
- Switching is fast. NO switch from user to protected mode

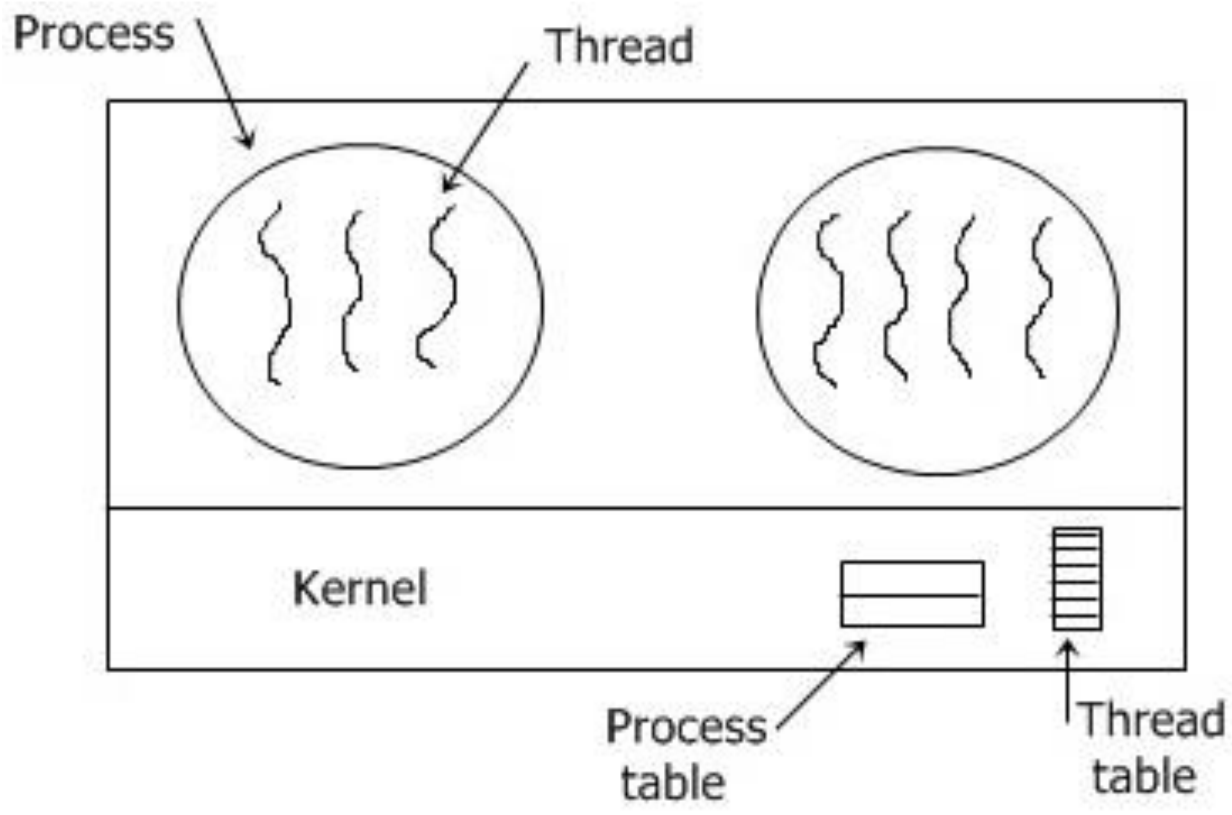
- Scheduling can be an issue
- Lack of coordination between kernel and threads
- If one thread invokes a system call, all threads need to wait



Kernel Level Threads

- Scheduler can decide to give more time to a process that large number of threads
- Since threads managed by kernel, no blocking on system calls

- Slow in comparison
- Overheads – scheduling threads apart from processes



References

- William Stallings, “Operating Systems: Internals and Design Principles”, 9th edition, Pearson Edu. Ltd., 2018
- Charles Crowley, “Operating Systems: A design-oriented approach”, TMH
- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau (University of Wisconsin-Madison), “Operating Systems: Three Easy Pieces”.

  URL: http://pages.cs.wisc.edu/~remzi/OSTEP/