UNIT - 2

PROCESS MANAGEMENT

Chapter 3: Process Concept

Objectives:
* To introduce the notion of a process - a program in execution, which forms the basis of all computation.
* To describe the various features of processes, including scheduling, creation and termination, and communication.
* To describe communication in client-server systems.

---

(1) Process Concept

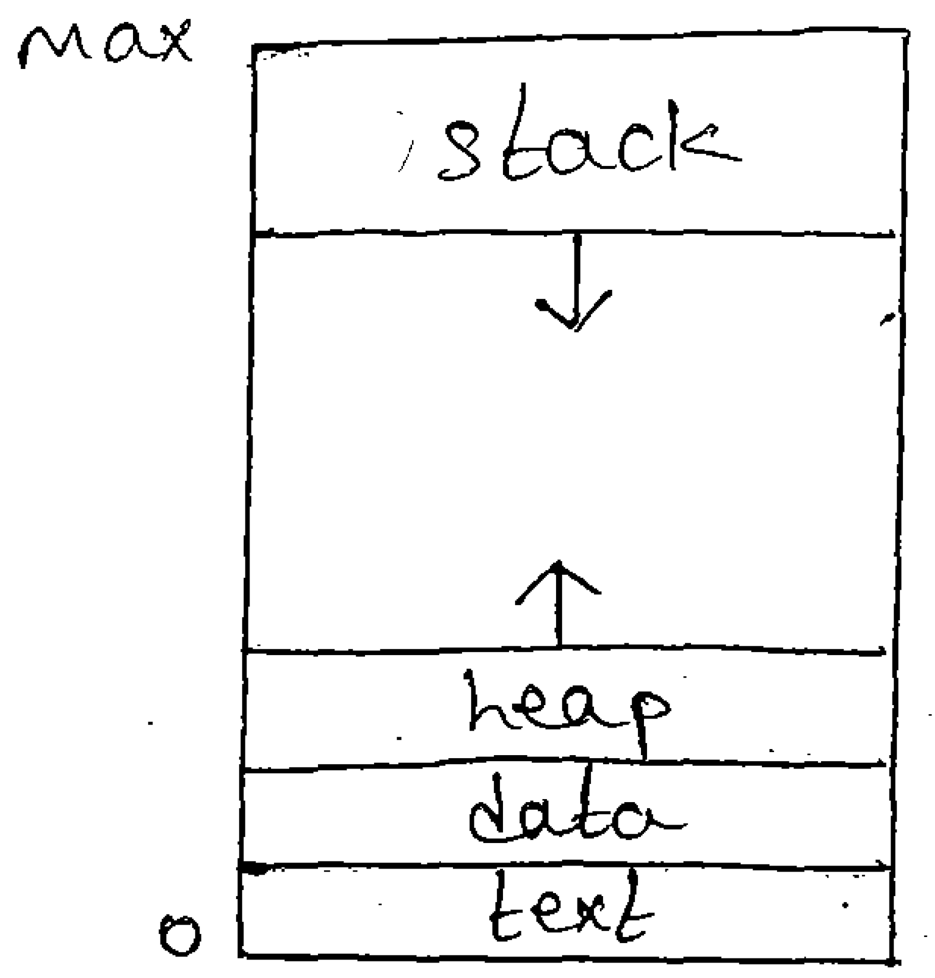
→ An OS executes a variety of programs:
* A batch system executes jobs
* A time-shared system has user programs (or) tasks.

→ Process - A program in execution; process execution must progress in sequential fashion.

→ A process includes:
* program counter
* stack
* data section

The process

→ Multiple parts:
* The program code, also called text section.
* Current activity including program counter, processor registers.
* Stack containing temporary data
  ⇒ Function parameters, return addresses, local variables
* Data section containing global variables.



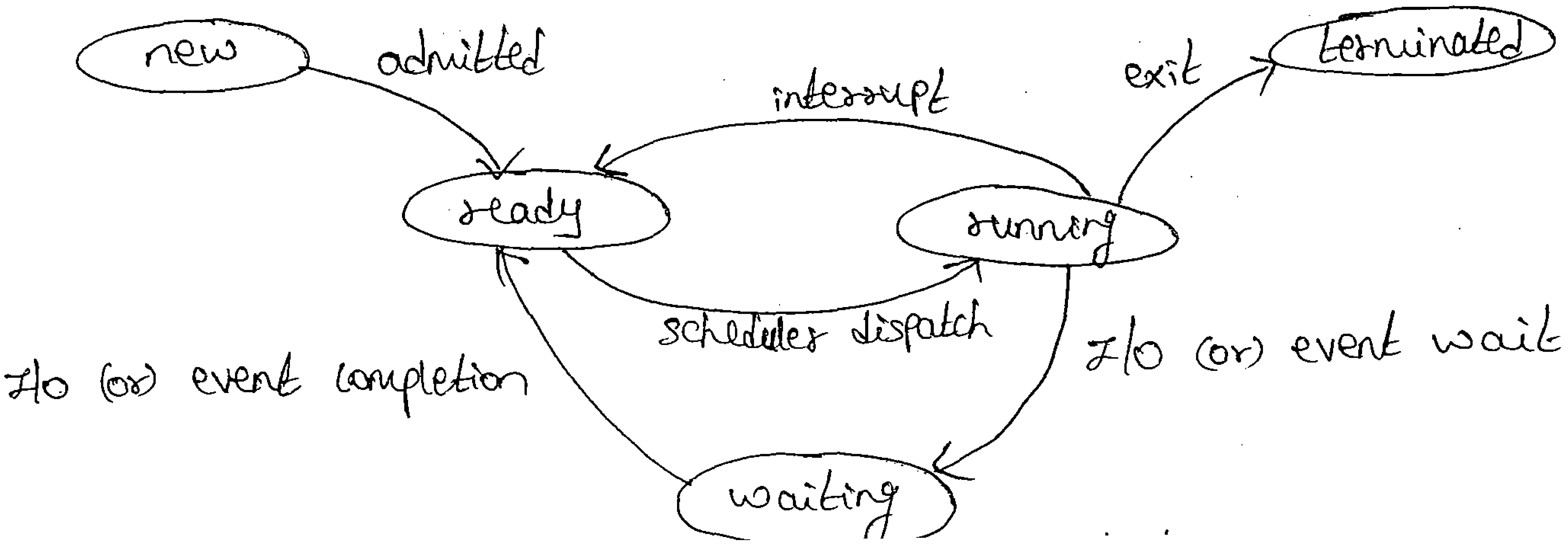
Process in memory

> Program is passive entity, process is active entity.
> Program becomes process when executable file loaded into memory.
> Execution of program started via GUI mouse clicks, command line entry of its name etc.
> One program can be several processes.
  * Consider multiple users executing the same program.

Process State

> As a process executes, it changes state.

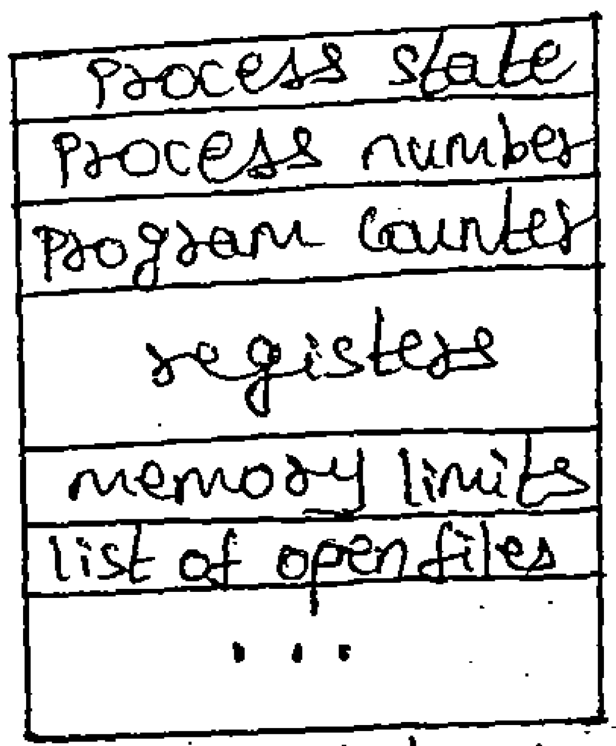
1) New - The process is being created.
2) Running - Instructions are being executed.
3) Waiting - The process is waiting for some event to occur.
4) Ready - The process is waiting to be assigned to a processor.
5) Terminated - The process has finished execution.



Process control block

→ Each process is represented in the OS by a process control block (PCB).

→ Also called a task control block.



Process control block (PCB)

→ PCB contains many pieces of information associated with a specific process:

- » Process state
- » program counter
- » CPU registers
- » CPU-scheduling information
- » Memory-management information
- » Accounting information
- » I/O status information

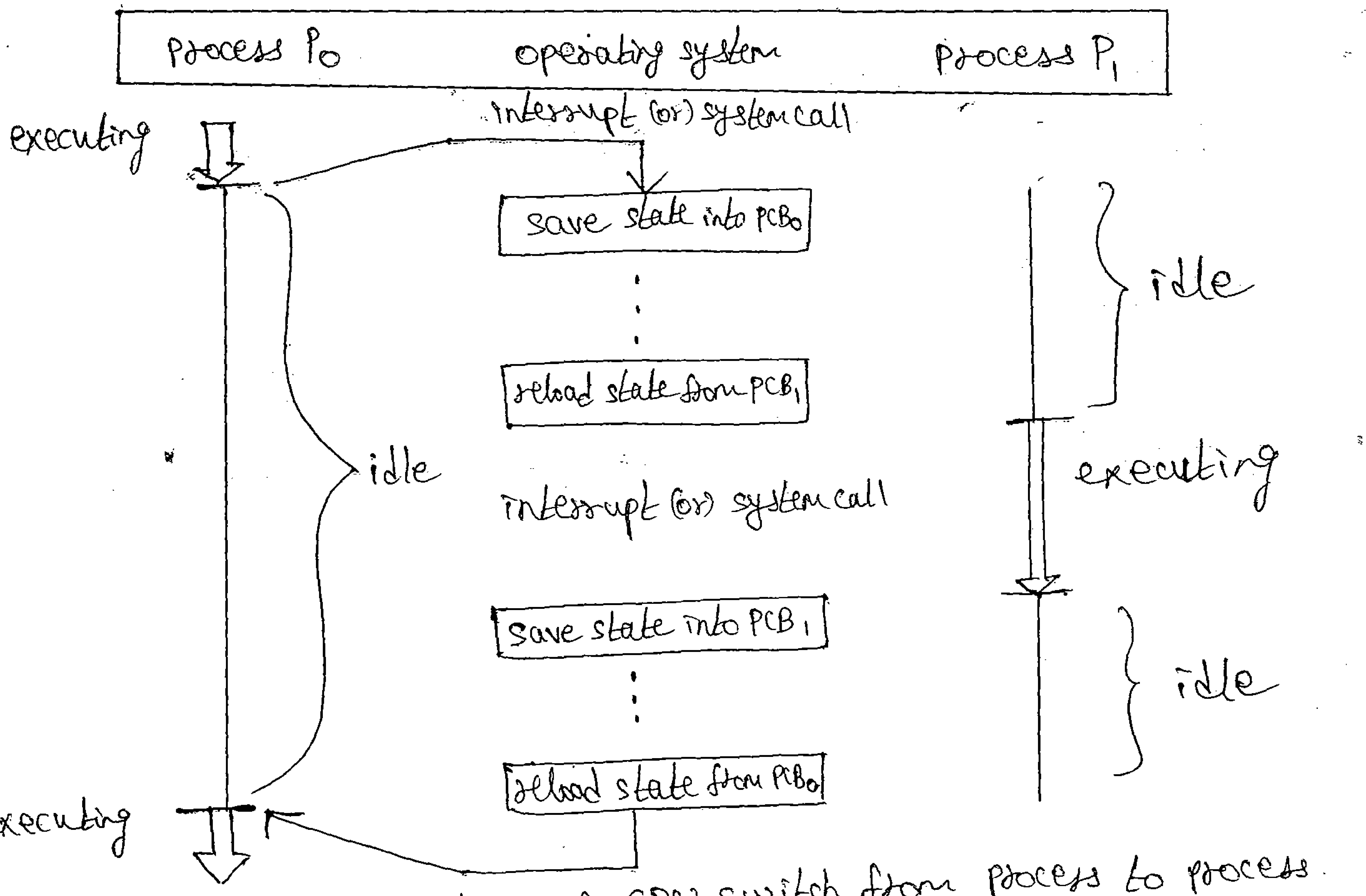


Diagram showing CPU switch from process to process.

(ii) Process Scheduling

→ Maximize CPU use, quickly switch processes onto CPU for time sharing.

→ Process scheduler selects among available processes for next execution on CPU.

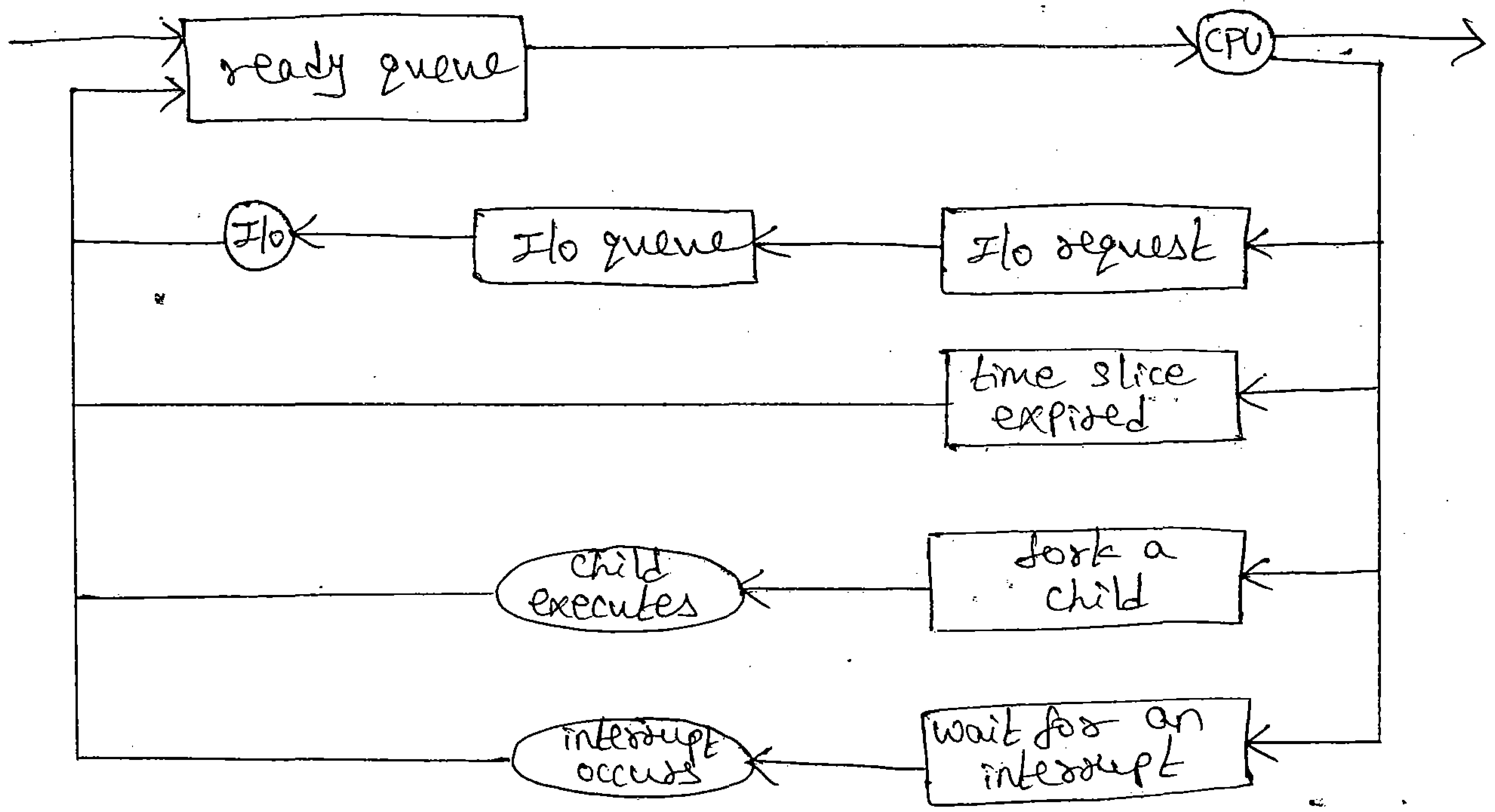
Scheduling Queues

* Job queue - set of all processes in the system.
* Ready queue - set of all processes residing in main memory, ready & waiting to execute. [Refer Figure 3.6]
* Device queues - set of processes waiting for an I/O device.

→ Processes migrate among the various queues.

→ A common representation of process scheduling is a queueing diagram.

* Each rectangular box represents a queue.
* Two types of queues are present: ready queue, set of device queues.
* Circles represent the resources that serve the queues.
* Arrows indicate the flow of processes in the system.



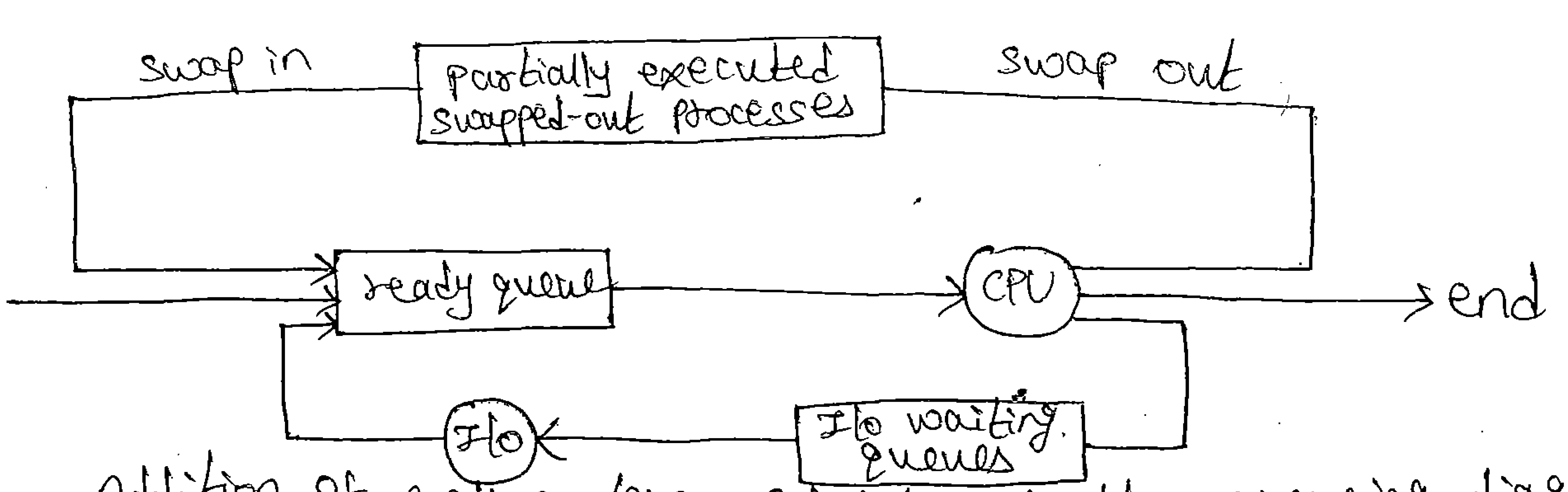
Queueing-diagram representation of process scheduling

→ A new process is initially put in a ready queue
→ It waits there until it is selected for execution (or) dispatched.
→ Once the process is allocated the CPU & is executing, one of the several events could occur:
  * The process could issue an I/O request and then be placed in an I/O queue.
  * The process could create a new subprocess and wait for the subprocess's termination.
  * The process could be removed forcibly from the CPU, as a result of an interrupt, & be put back in the ready queue.

Schedulers

→ Long-term scheduler (or job scheduler) - selects which processes should be brought into the ready queue.
→ Short-term scheduler (or CPU scheduler) - selects which process should be executed next and allocates CPU.
→ Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
→ Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
→ The long-term scheduler controls the degree of multiprogramming (the number of processes in memory).
→ 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.

→ Time-sharing systems, may introduce an additional intermediate level of scheduling - Medium-term scheduler.
  * swapping.



Addition of medium-term scheduling to the queueing diagram

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.

→ Context of a process represented in the PCB.

→ Context-switch time is overhead; the system does no useful work while switching
  * The more complex the OS & the PCB → longer the context switch.

→ Time dependent on hardware support
  * some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once.

Ⓡ Operations on Processes

Process Creation

→ The creating process is called a Parent process.

→ parent process create children processes, which in turn create other processes, forming a tree of processes.

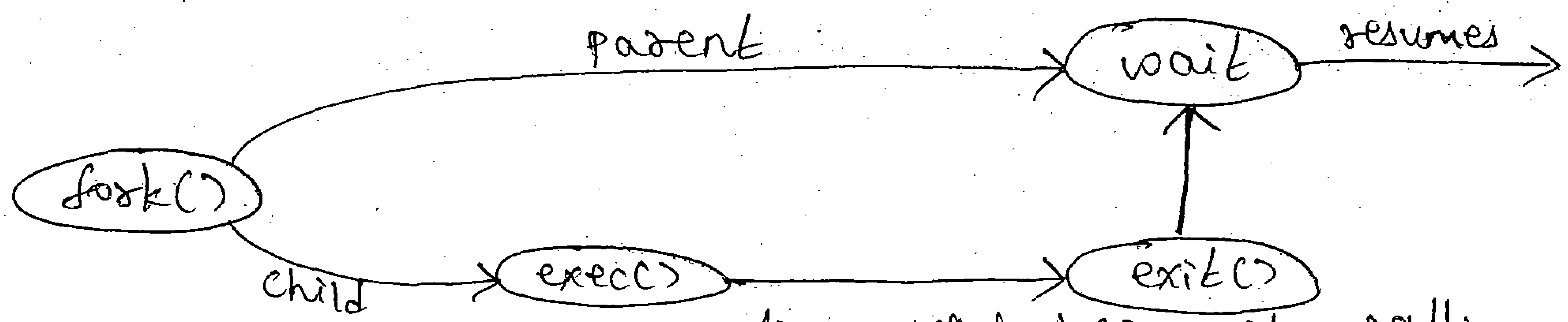
→ Generally, process identified and managed via a process identifier (pid).

→ Resource sharing:
  * parent and children share all resources
  * children share subset of parent's resources.

→ Execution: When a process creates a new process:-
  (i) The parent continues to execute concurrently with its children.
  (ii) The parent waits until some (or) all of its children have terminated.

→ Also two possibilities in terms of address space of the new process:
  (i) The child process is a duplicate of the parent process.
  (ii) The child process has a new program loaded into it.

→ UNIX examples:
  * fork () system call creates new process.
  * exec() system call used after a fork() to replace the process' memory space with a new program.



Process creation using fork() system call.

Process Termination

→ Process executes last statement and asks the OS to delete it by using exit() system call.
→ output data from child to parent (via wait() system call).
→ process' resources are deallocated by OS.

→ parent may terminate execution of children processes (via abort() system call).
  * Child has exceeded allocated resources
  * task assigned to child is no longer required.
  * If parent is exiting
    » some OS do not allow child to continue if its parent terminates.
      - All children terminated: Cascading termination.

(7) Interprocess Communication

→ A process is independent if it cannot affect (or) be affected by the other processes executing in the system.

→ Cooperating process can affect (or) be affected by other processes, including sharing data.

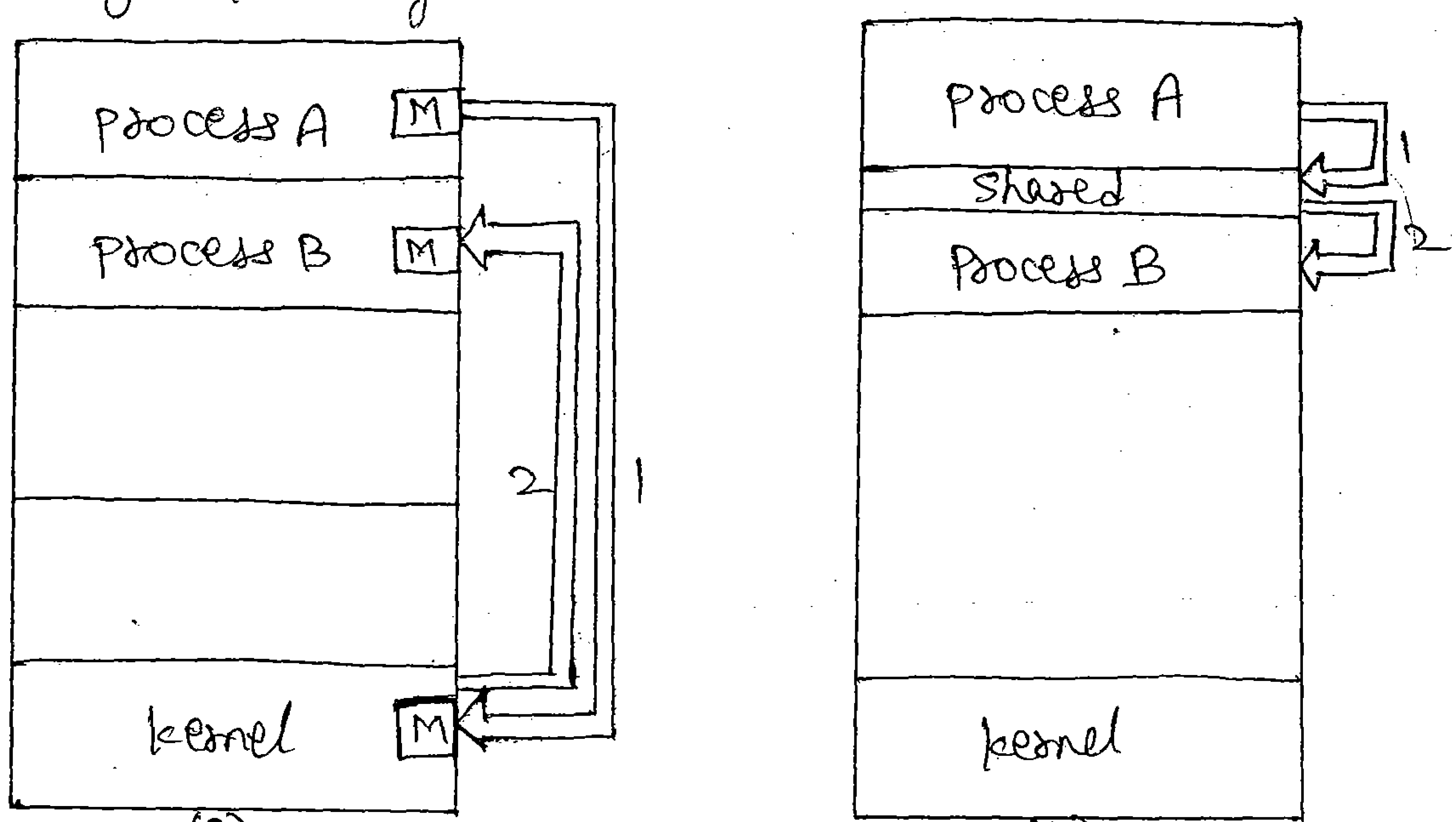
→ Reasons for providing an environment that allows process cooperation:

* Information sharing
* Computation speedup
* Modularity
* Convenience

→ Cooperating processes need interprocess communication (IPC)
↳ Mechanism that will allow them to exchange data and information

→ Two models of IPC:

(i) Shared Memory
(ii) Message passing



Communications models. (a) Message Passing. (b) Shared memory.

Shared-Memory Systems

→ A region of memory that is shared by cooperating processes is established.

→ Processes can then exchange information by reading & writing

→ Producer-Consumer problem

* Paradigm for cooperating processes.
* Producer process produces information that is consumed by a consumer process.
* Example: A compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.
* One solution to the producer-consumer problem uses shared memory.
* Two types of buffers:
  » Unbounded-buffer places no practical limit on the size of the buffer.
  » Bounded-buffer assumes that there is a fixed buffer size.

```
#define BUFFER_SIZE 10
typedef struct {
    ...
}item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

* Solution is correct, but can only use BUFFER_SIZE -1 elements.

→ The code for the producer and consumer processes:

```
item nextProduced;
while (true) {
    /* Produce an item in nextProduced */
    while ((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
```

```
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

The producer process

```
item nextConsumed;
while (true) {
    while (in == out)
        ; // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /* consume the item in nextConsumed */
}
```

The Consumer Process

Message-Passing Systems

> Mechanism for processes to communicate and to synchronize their actions in distributed environment.

> Message system - processes communicate with each other without resorting to shared variables/ same address space.

> A message-passing facility provides atleast two operations:
  * send (message) - message size fixed (or) variable.
  * receive (message) Ex:- chat program used on the WWW.

> If processes P and Q want to communicate, they need to:
  * Establish a communication link between them.
  * Exchange messages via send/receive.

> Implementation of communication link:
  * physical (Ex:- shared memory, hardware bus)
  * logical (Ex:- logical properties)

→ Methods for logically implementing a link and the send()/receive() operations:

  » Direct (or) indirect communication
  » Synchronous (or) asynchronous communication.
  » Automatic (or) explicit buffering.

→ Naming

* Under direct communication, each process that wants to communicate must explicitly name the recipient (or) sender of the communication.

  » send (P, message) - Send a message to process P.
  » receive (Q, message) - Receive a message from process Q.

* Properties of communication link:

  » Links are established automatically
  » A link is associated with exactly one pair of communicating processes.
  » Between each pair there exists exactly one link.
  » The link may be unidirectional, but is usually bi-directional.

* With indirect communication, messages are directed and received from mailboxes (also referred to as ports)

  » Each mailbox has a unique id.
  » Processes can communicate only if they share a mailbox.
  » send (A, message) – send a message to mailbox A.
  » receive (A, message) – Receive a message from mailbox A.

* Properties of communication link:

  » Link established only if processes share a common mailbox.
  » A link may be associated with many processes.

» Each pair of processes may share several communicatio links.

» Link may be unidirectional (or) bi-directional.

* Operations:

» Create a new mailbox.

» send and receive messages through the mailbox.

» Delete a mailbox.

* Mailbox sharing:

» P_1, P_2 & P_3 share mailbox A

» P_1, sends; P_2 and P_3 receive

» which process will receive the message sent by P_1?

* Solutions:

» Allow a link to be associated with at most two processes.

» Allow only one process at a time to execute a receive operation.

» Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

→ Synchronization

* message passing may be either blocking (or) non-blocking.

* Blocking is considered as synchronous.

» Blocking send has the sender block until the message is received.

» Blocking receive has the receiver block until a message is available.

* Non-blocking is considered as Asynchronous

» Non-blocking send has the sender send the message and continue.

→ Non-blocking receive has the receiver receive a valid message (or) null.

→ Buffering

* Queue of messages attached to the link; implemented in one of three ways:

(i) Zero capacity - queue has a maximum length of '0'. sender must wait (block) for receiver.

(ii) Bounded capacity - queue has finite length 'n'. Sender must wait (block) if link is full.

(iii) Unbounded capacity - queue's length is potentially infinite. The sender never blocks.

Chapter 4: Multithreaded Programming

Objectives: * To introduce the notion of a thread - a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems.

* To discuss the APIs for the Pthreads, Win32 and Java thread libraries.

* To examine issues related to multithreaded programming.

Overview

→ A thread is a basic unit of CPU utilization.

→ A thread is a single sequence stream within in a process.

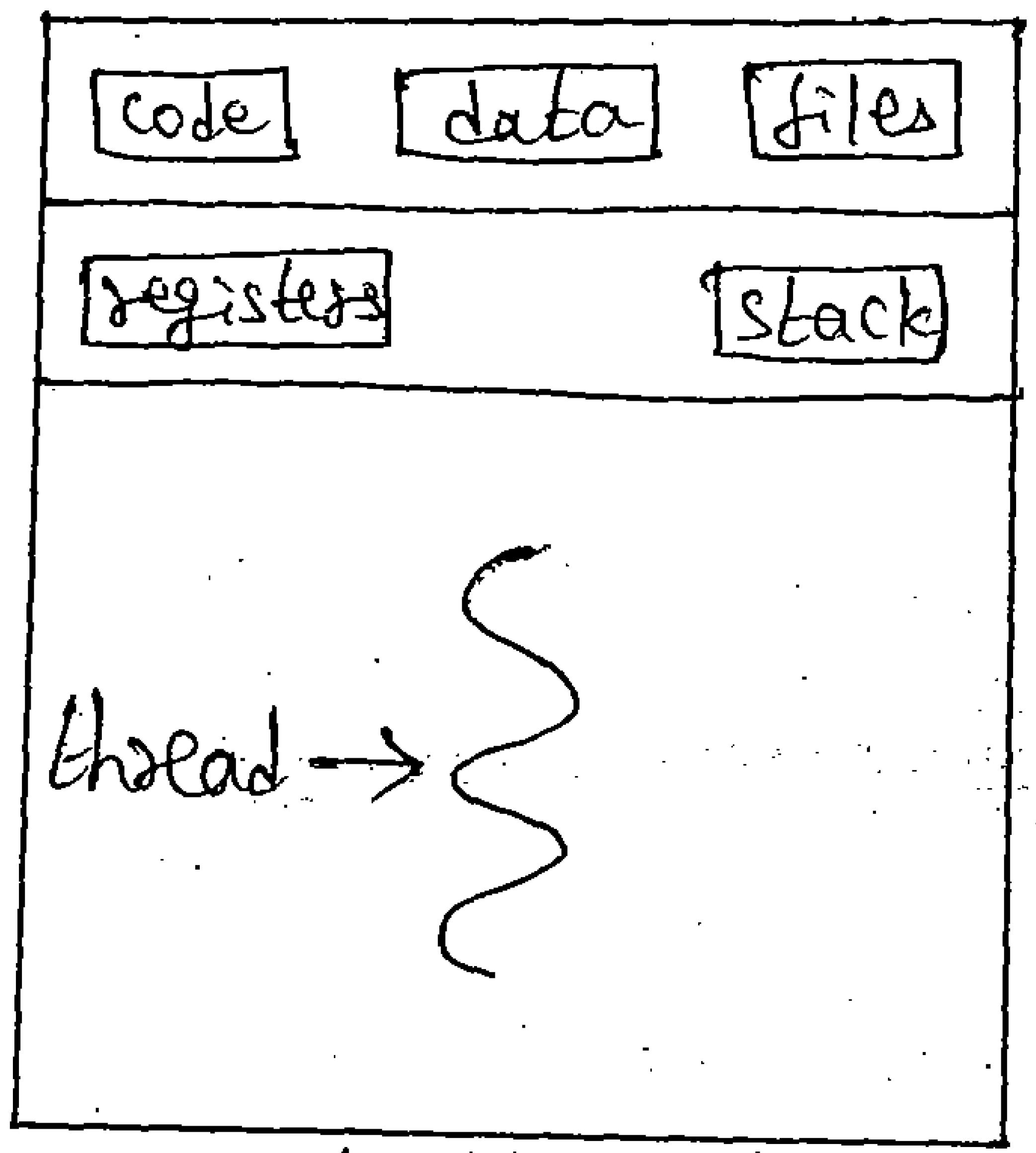
→ A traditional (or heavyweight) process has a single thread of control.

→ Multiple tasks with the application can be implemented by separate threads:

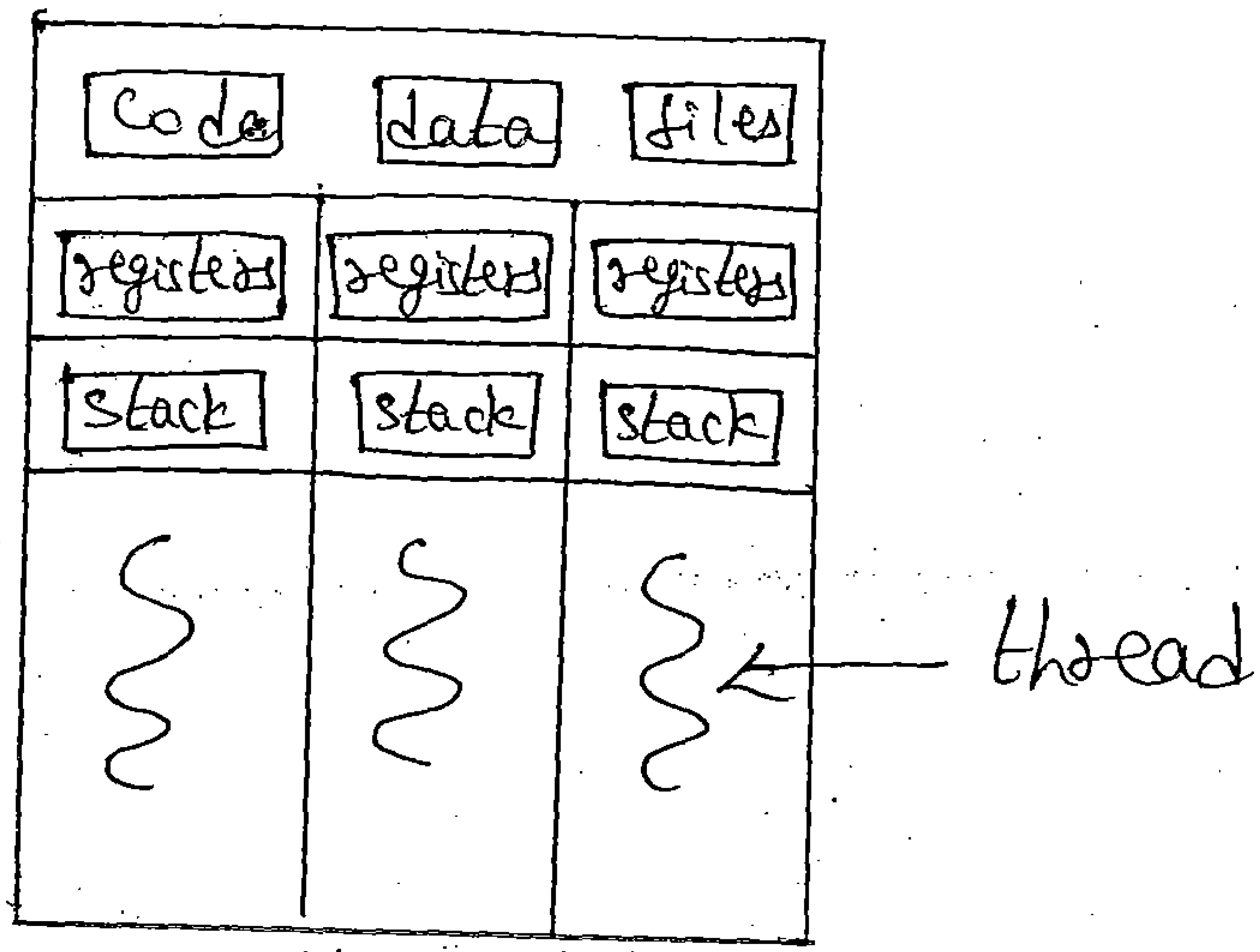
* Update display
* Fetch data
* Spell checking
* ... network request.

→ Process creation is heavy-weight while thread creation is light-weight.

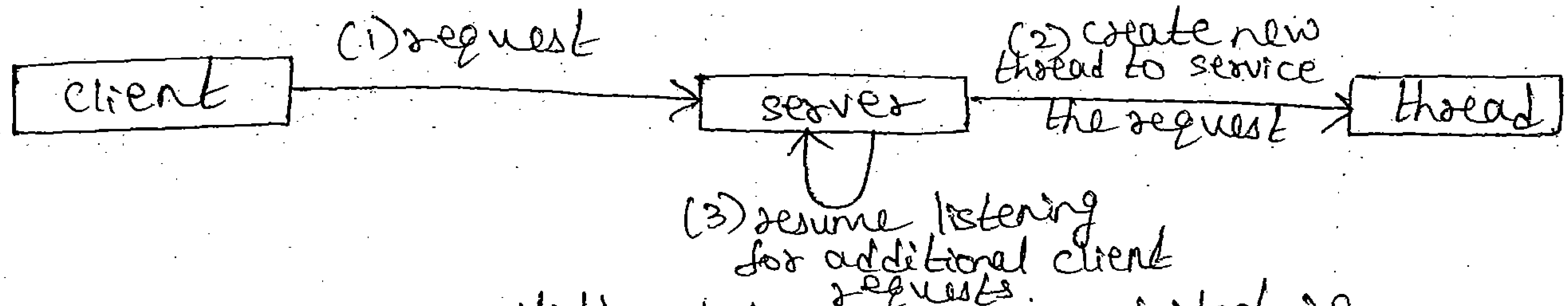
→ kernels are generally multithreaded.



Single-threaded process



Multi-threaded process



Multithreaded server architecture

Benefits

* Responsiveness - multithreading an interactive application may allow a program to continue running even if part of it is blocked (or) is performing a lengthy operation, thereby increasing responsiveness to the user.

* Resource sharing - Processes may only share resources through techniques such as shared memory (or) message passing.

* Economy - Allocating memory and resources for process creation is costly. More economical to create & context-switch threads

* Scalability - The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors.

Multicore Programming

single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	T_2	T_3	...

time →

multicore

core 1	T_1	T_3	T_1	T_3	T_1	...
core 2	T_2	T_4	T_2	T_4	T_2	...

time →

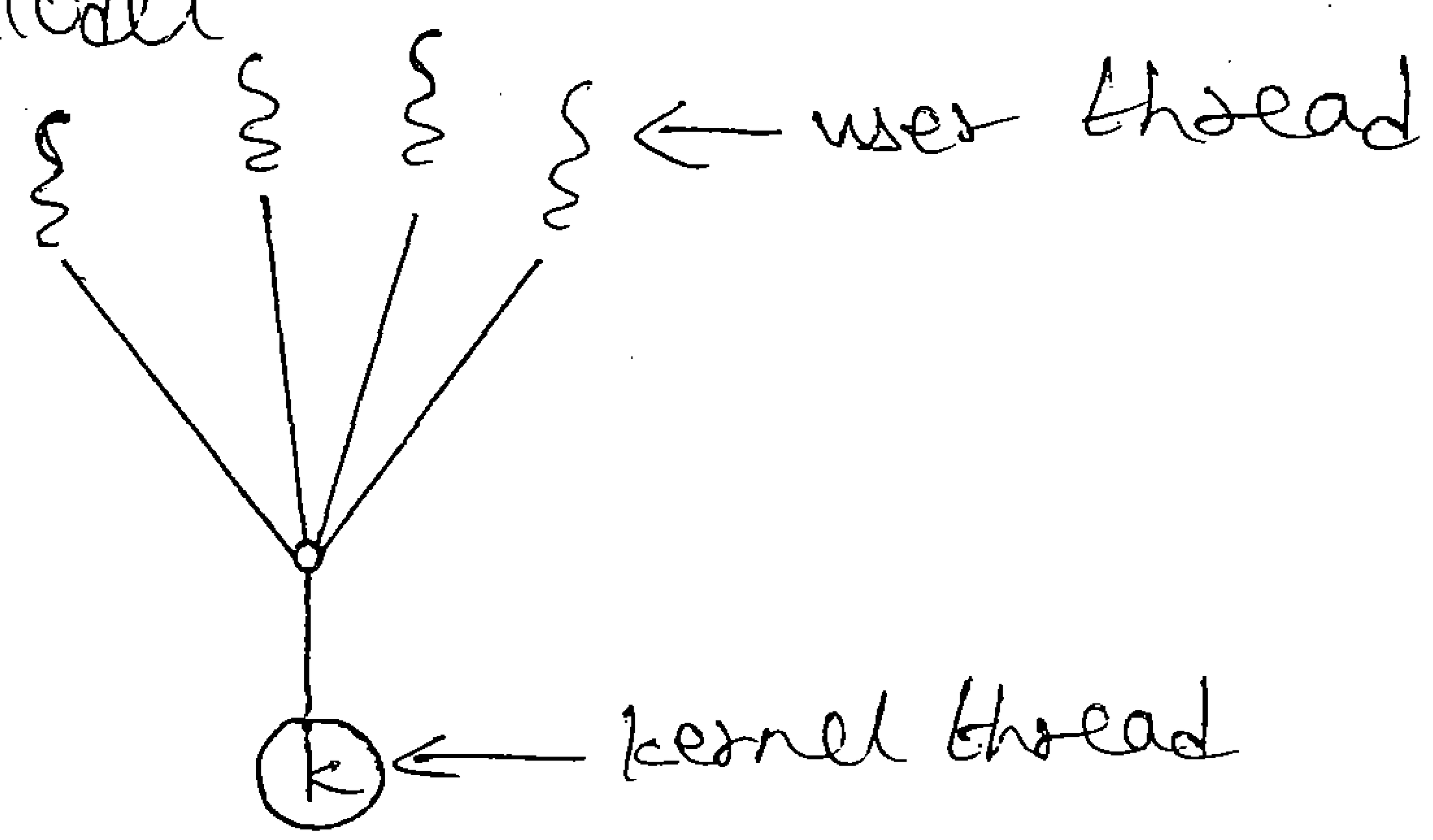
* Multicore systems putting pressure on programmers, challenges include:
  (i) Dividing activities
  (ii) Balance
  (iii) Data splitting
  (iv) Data dependency
  (v) Testing and debugging

Multithreading Models

→ Support for threads may be provided either at the user level, for user threads; or by the kernel, for kernel threads.

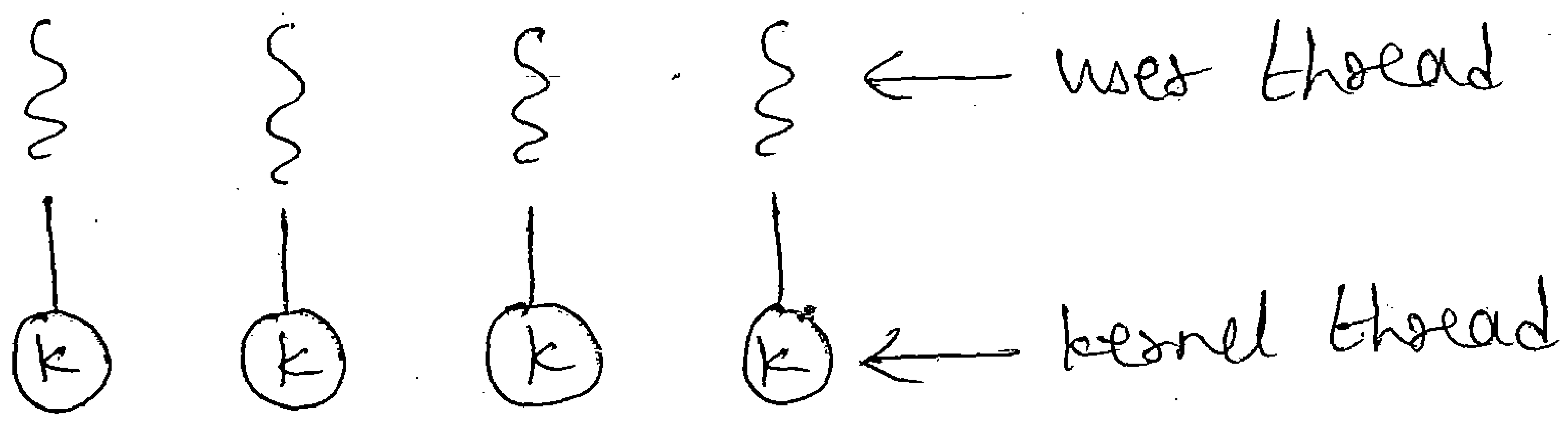
→ Kernel Threads examples: Windows XP, Linux, Mac OS X, Solaris and Tru64 UNIX (formerly Digital UNIX) - support kernel threads.

Many-to-one Model



→ Many user-level threads mapped to single kernel thread.
→ Examples: Solaris Green Threads, GNU Portable Threads.

One-to-One Model

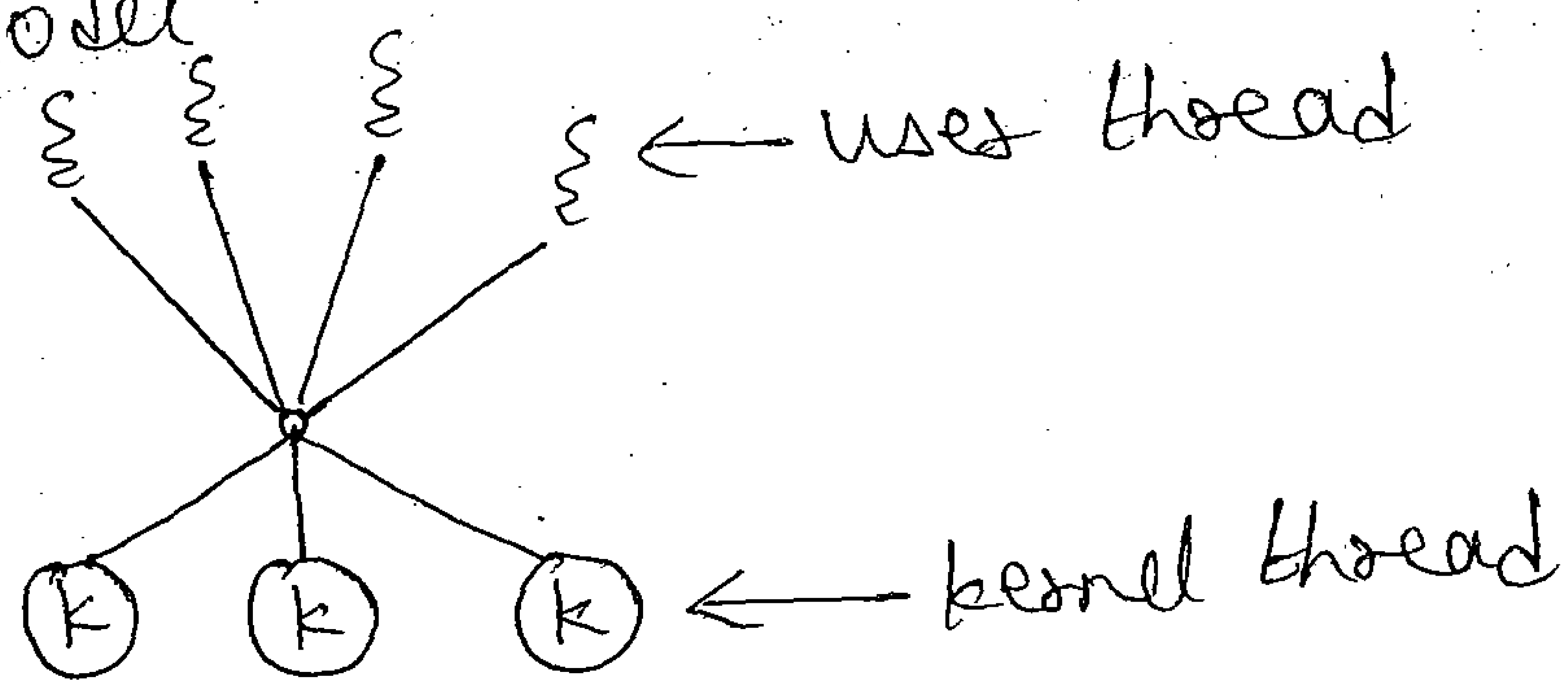


→ Each user-level thread maps to kernel thread.

→ It provides more concurrency than many-to-one model by allowing another thread to run when a thread makes a blocking system call.

→ Examples: Windows NT/XP/2000, Linux, Solaris 9 & later.

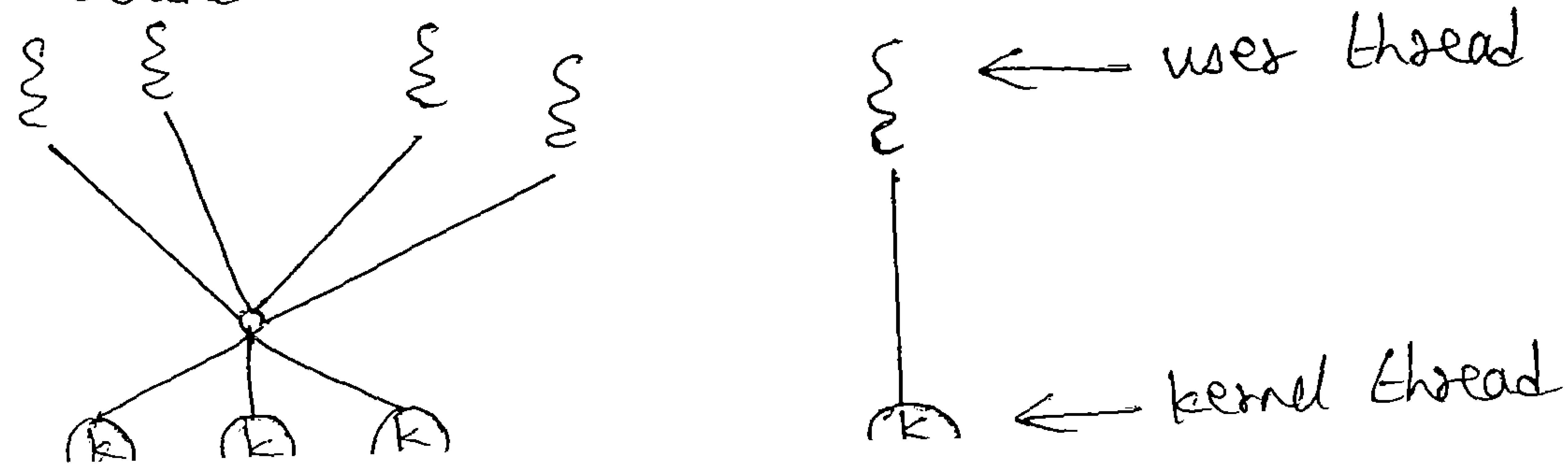
Many-to-Many Model



→ Multiplexes ~~many~~ many user-level threads to a smaller (or) equal number of kernel threads.

→ Allows the OS to create a sufficient number of kernel threads.

→ One popular variation on the many-to-many model still multiplexes many user-level threads to a smaller (or) equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread — Two-level model



→ Solaris prior to version 9.

→ Two-level model is supported by OSs: IRIX, HP-UX and Tru64 Unix.

→ Windows NT/2000 with the ThreadFiber package.

Thread Libraries

→ A thread library provides the programmer with an API for creating and managing threads.

→ Two primary ways of implementing:
- * Library entirely in user space.
- * kernel-level library supported by the OS.

→ Three main thread libraries:
1) POSIX Pthreads
2) Win32
3) Java

Pthreads

→ May be provided either as user-level (or) kernel-level.

→ A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.

→ API specifies behavior of the thread library, implementation is up to development of the library.

→ Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Win32 Threads

→ The technique for creating threads using the Win32 thread library is similar to the Pthreads technique in several ways.

Java Threads

→ Threads are the fundamental model of program executi in a Java program.

→ Java threads are managed by the JVM.

→ Two techniques for creating threads in a Java program.

1) Create a new class that is derived from the Thread class and to override its run() method.

2) Define a class that implements the Runnable interface. The Runnable interface is defined as follows:

```
public interface Runnable
{
    public abstract void run();
}
```

→ Creating a Thread object does not specifically create the new thread.

→ start() method that creates the new thread.

→ Calling the start() method for the new object does two things:

(i) It allocates memory and initializes a new thread in the JVM.

(ii) It calls the run() method making the thread eligible to be run by the JVM.

---

7) Threading Issues

The fork() and exec() System Calls

→ The semantics of the fork() and exec() system calls change in a multithreaded program.

→ If one thread in a program calls fork(), does the new process duplicate all threads, (or) is the new process single-threaded?

* Some UNIX systems have chosen to have two versions of fork().

→ If a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process — including all threads.

Cancellation

→ Thread cancellation is the task of terminating a thread before it has completed.

→ A thread that is to be canceled is often referred to as: Target Thread.

→ Cancellation of target thread may occur in two different scenarios:

1) Asynchronous cancellation – One thread immediately terminates the target thread.
2) Deferred Cancellation – The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

Signal Handling

→ A signal is used in UNIX systems to notify a process that a particular event has occurred.

→ A signal handler is used to process signals. All signals follows

(i) A signal is generated by the occurrence of a particular event.
(ii) A generated signal is delivered to a process.
(iii) Once delivered, the signal must be handled.

→ A signal may be handled by one of two possible handlers:

1) A default signal handler.
2) A user-defined signal handler.

→ Options:

* Deliver the signal to the thread to which the signal applies.
* Deliver the signal to every thread in the process.
* Deliver the signal to certain threads in the process.
* Assign a specific thread to receive all signals for the process.

→ Windows does not explicitly provide support for signals, they can be emulated using asynchronous procedure calls (APCs).

Thread Pools

→ Create a number of threads in a pool where they await work.

→ Thread pools offer these benefits:

1) Servicing a request with an existing thread is usually faster than waiting to create a thread.

2) A thread pool limits the number of threads that exist at any one point.
This is particularly important on systems that cannot support a large number of concurrent threads.

= Thread-Specific Data

· Allows each thread to have its own copy of data.

Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

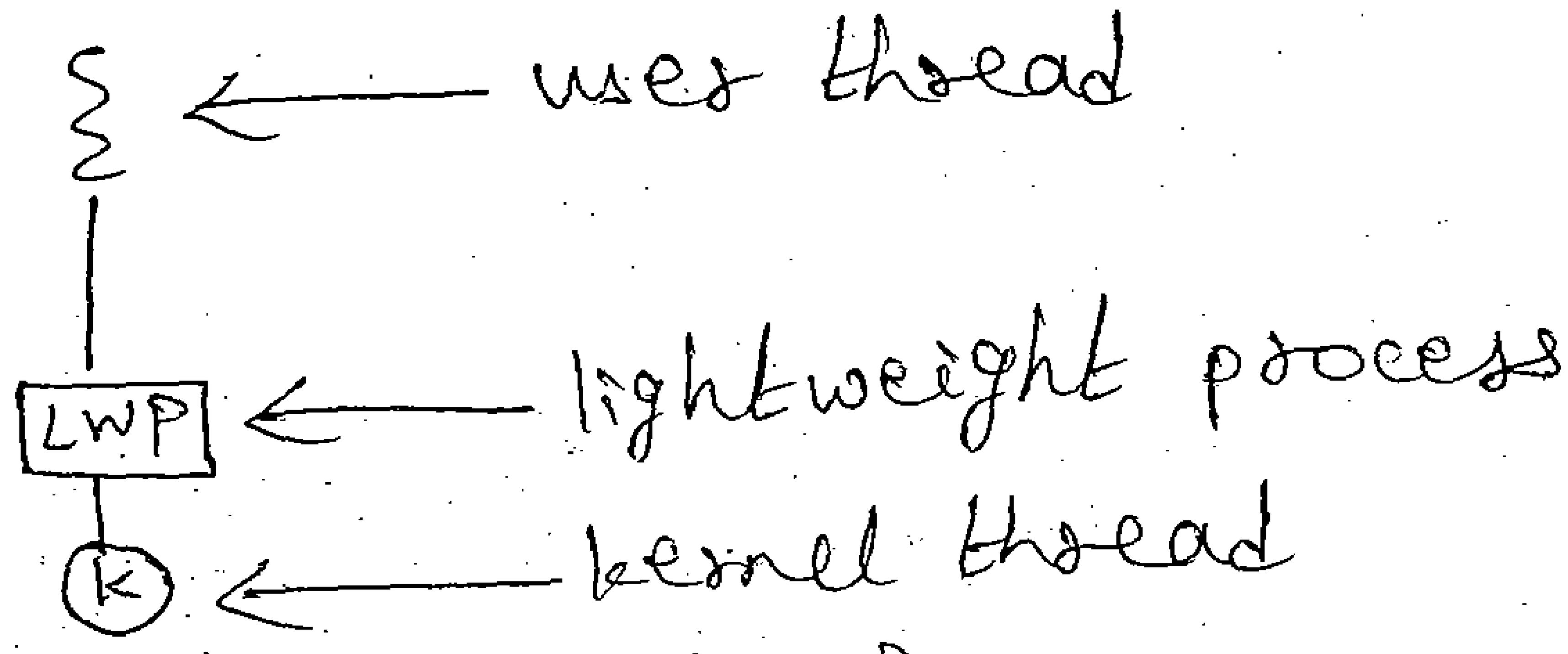
· Most thread libraries – including Win32 and Pthreads provide some form of

Scheduler Activations

→ Both many-to-many and two-level models require communication to maintain the appropriate number of kernel threads allocated to the application.

→ Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library.

→ This communication allows an application to maintain the correct number of kernel threads.



Lightweight process (LWP)

Chapter 5: Process Scheduling

Objectives:
* To introduce CPU scheduling, which is the basis for multiprogrammed operating systems.
* To describe various CPU-scheduling algorithms
* To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system.

---

(1) Basic Concepts.

→ In a single-processor system, only one process can run at a time.

→ The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

→ The problem of determining when processors should be assigned and to which processes is called process scheduling (or) CPU Scheduling.

> When more than one process is runnable, the OS must decide which one first. The part of the OS concerned with this decision is called the scheduler.

* Algorithm it uses is called the scheduling algorithm.

CPU-I/O Burst Cycle

> Process execution consists of a cycle of CPU execution and I/O wait.

> Processes alternate between these two states:

(i) process execution begins with a CPU burst.

(ii) followed by an I/O burst.

<u>Alternating sequence of CPU & I/O bursts</u>

⋮

load store
add store
read from file — CPU burst

[wait for I/O] — I/O burst

store increment
index
write to file — CPU burst

[wait for I/O] — I/O burst

load store
add store
read from file — CPU burst

[wait for I/O] — I/O burst

⋮

CPU Scheduler

> selects from among the processes in ready queue, and allocates the CPU to one of them.

* Queue may be ordered in various ways.

> A ready queue can be implemented as a FIFO queue, a priority queue, a tree (or) simply an unordered linked list.

Preemptive Scheduling

→ CPU-scheduling decisions under four circumstances:

1) When a process switches from the running state to the waiting state.

2) When a process switches from the running state to the ready state.

3) When a process switches from the waiting state to the ready state.

4) when a process terminates.

→ Scheduling under 1) and 4) is nonpreemptive (or) cooperative

→ All other scheduling is preemptive.

* Consider access to shared data.
* Consider preemption while in kernel mode
* Consider interrupts occurring during crucial OS activities

Dispatcher

→ Dispatcher module gives control of the CPU to the process selected by the short-term scheduler, this involves:

* Switching context
* switching to user mode
* Jumping to the proper location in the user program to restart that program.

→ Dispatch latency - time it takes for the dispatcher to stop one process and start another running.

Scheduling Criteria

→ The Criteria include the following:

(i) CPU utilization - keep the CPU as busy as possible.

(ii) Throughput - Number of processes that complete their execution per time unit.

(iii) Turnaround time - Amount of time to execute a particular process.

(iv) Waiting time - Amount of time a process has been waiting in the ready queue.

(v) Response time - Amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithms

→ scheduling algorithm Optimization Criteria

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

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:

P_1	P_2	P_3

0   24   27   30

* Gantt chart - A bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes.
* Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$ milliseconds.
* Average waiting time: $(0+24+27)/3 = 17$ milliseconds.

* Turnaround time for $P_1 = 24$; $P_2 = 27$; $P_3 = 30$ milliseconds
* Average Turnaround time: $(24+27+30)/3 = 27$ milliseconds

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

P_2	P_3	P_1

0   3   6   30

* Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$ milliseconds.
* Average waiting time: $(6+0+3)/3 = 3$ milliseconds.

* Turnaround time for $P_1 = 30$; $P_2 = 3$; $P_3 = 6$ milliseconds.
* Average Turnaround time: $(30+3+6)/3 = 13$ milliseconds

→ Much better than previous case.

→ Convoy effect - short process behind long process.
  * Consider one CPU-bound and many I/O-bound processes.

Shortest-Job-First (SJF) Scheduling

→ Also called as Shortest-Process-Next (SPN).

→ Non-preemptive discipline

→ Associate with each process the length of its next CPU burst.

* Use these lengths to schedule the process with the shortest time.

→ SJF is optimal - gives minimum average waiting time for a given set of processes.

* The difficulty is knowing the length of the next CPU request.

→ Two schemes:

* Non-preemptive - once CPU given to the process it cannot be preempted until completes its CPU burst.

* Preemptive - if a new process arrives with CPU burst length less than remaining time of current executing process, preempt.
  - → This scheme is known as Shortest-Remaining-Time-First (SRTF) Scheduling.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

* SJF Scheduling, Gantt Chart:

P_4	P_1	P_3	P_2

* Waiting time for $P_1 = 3$; $P_2 = 16$; $P_3 = 9$; $P_4 = 0$ milliseconds.
* Average waiting time : $(3+16+9+0)/4 = 7$ milliseconds.

* Turnaround time for $P_1 = 9$; $P_2 = 24$; $P_3 = 16$; $P_4 = 3$ millisecon
* Average turnaround time : $(9+24+16+3)/4 = 13$ milliseconds.

→ The next CPU burst is generally predicted as an Exponentia Average of the measured lengths of previous CPU bursts.

→ Exponential average formula:
(i) t_n = length of the nth CPU burst.
(ii) τ_{n+1} = predicted value for the next CPU burst.
(iii) $\alpha, 0 \leq \alpha \leq 1$ define,

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n.$$

→ Example for shortest-remaining-time-first (SRTF) scheduling.

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

* Preemptive SJF Gantt chart:

P_1	P_2	P_4	P_1	P_3
0–1	1–5	5–10	10–17	17–26

* Waiting time for $P_1 = (10-1)$; $P_2 = (1-1)$; $P_3 = (17-2)$; $P_4 = (5-3)$ milliseconds
* Average waiting time : $[(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4$
$= 6.5$ milliseconds.
* Non-preemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

* Turnaround time for $P_1 = 17$; $P_2 = 5$; $P_3 = 26$; $P_4 = 10$ milliseconds.
* Average Turnaround time : $(17+5+26+10)/4 = 58/4 = 14.5$ milliseconds.

Priority Scheduling

→ SJF algorithm is a special case of the general priority scheduling algorithm.

→ A priority number (integer) is associated with each process.

→ CPU is allocated to the process with the highest priority. (smallest integer = highest priority)

→ Can be either preemptive (or) non-preemptive.

→ A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

→ A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

Major problem with priority scheduling is Indefinite blocking (or) Starvation.

* Low priority processes may never execute

Solution – Aging: as time progresses increase the priority of the process.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

* Priority scheduling Gantt chart:

P2	P5	P1	P3	P4

0 1 6 16 18 19

* Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 16$; $P_4 = 18$; $P_5 = 1$ milliseconds.
* Average waiting time: $(6+0+16+18+1)/5 = 41/5 = 8.2$ milliseconds.
* Turnaround time for $P_1 = 16$; $P_2 = 1$; $P_3 = 18$; $P_4 = 19$; $P_5 = 6$ milliseconds
* Average Turnaround time: $(16+1+18+19+6)/5 = 60/5 = 12$ milliseconds

---

Round-Robin Scheduling

→ Designed especially for time-sharing systems.

→ Similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.

→ Small unit of time called time quantum (q).

→ After this time has elapsed, the process is preempted and added to the end of the ready queue.

→ If there are 'n' processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once.

  * No process waits more than $(n-1)q$ time units.

→ Timer interrupts every quantum to schedule next process.

→ Performance:

  * q large ⇒ FIFO
  * q small ⇒ q must be large with respect to context switch, otherwise overhead is too high.

→ Example of RR with Time Quantum, $q = 4$.

Process	Burst Time
P_1	24
P_2	3
P_3	3

* The Gantt chart is:

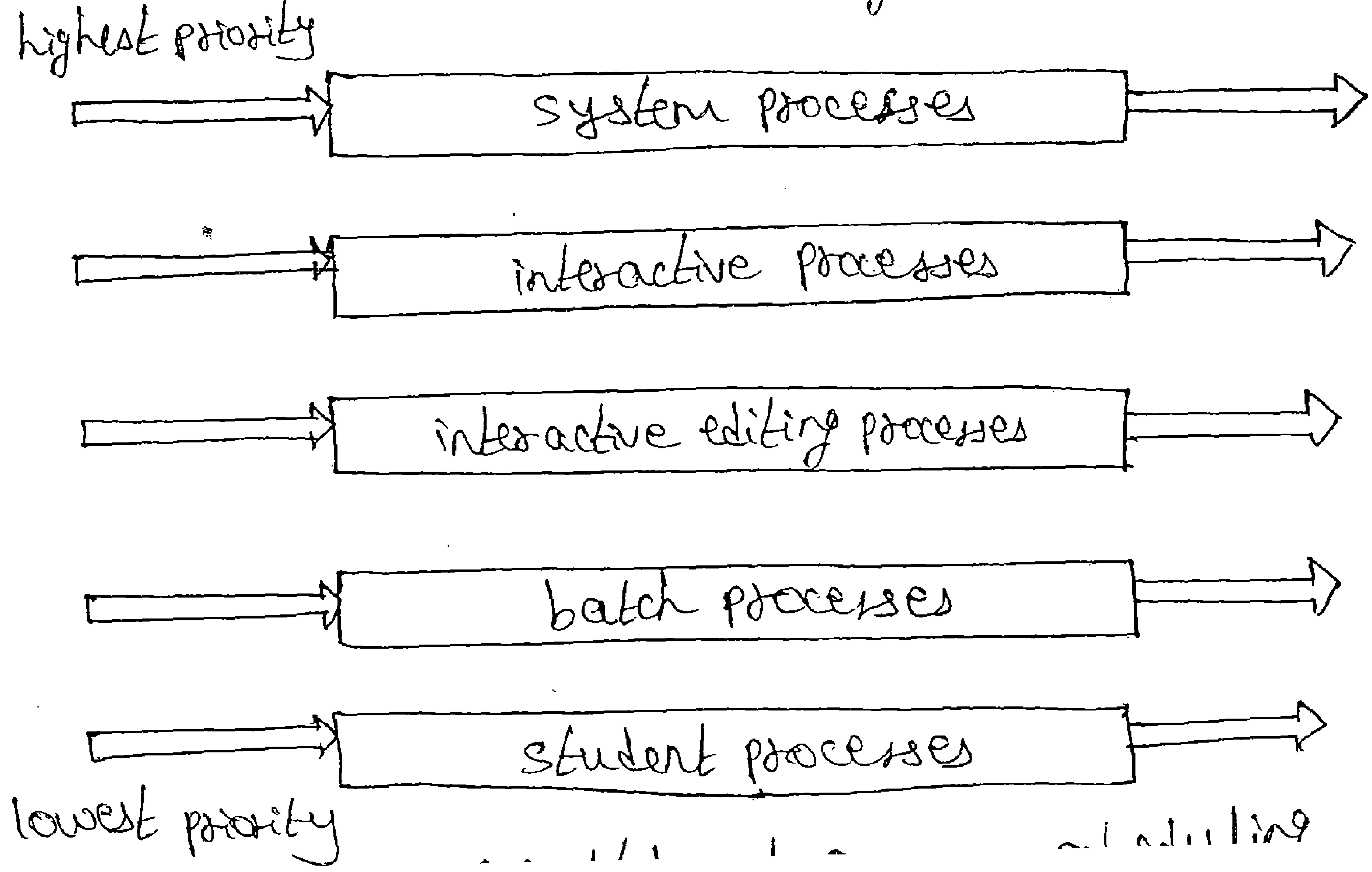
P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

* Waiting time for $P_1 = 0 + (10-4)$; $P_2 = 4$; $P_3 = 7$ milliseconds.
* Average waiting time : $[(10-4) + 4 + 7]/3 = 17/3 = 5.66$ milliseconds
* Turnaround time for $P_1 = 30$; $P_2 = 7$; $P_3 = 10$ milliseconds.
* Average Turnaround time : $(30 + 7 + 10)/3 = 47/3 = 15.66$ milliseconds

→ Typically, higher average turnaround than SJF, but better response

* Multilevel Queue scheduling



→ Ready queue is partitioned into separate queues, example.
* Foreground (interactive) processes
* Background (batch) processes.

→ Process permanently in a given queue

→ Each queue has its own scheduling algorithm:
* foreground - RR
* Background - FCFS

→ Scheduling must be done between the queues:
* Fixed priority scheduling - possibility of starvation
* Time slice - each queue gets a certain amount of CPU time which it can schedule amongst its processes i.e., 80% to foreground in RR.
* 20% to background in FCFS.

Multilevel Feedback Queue Scheduling

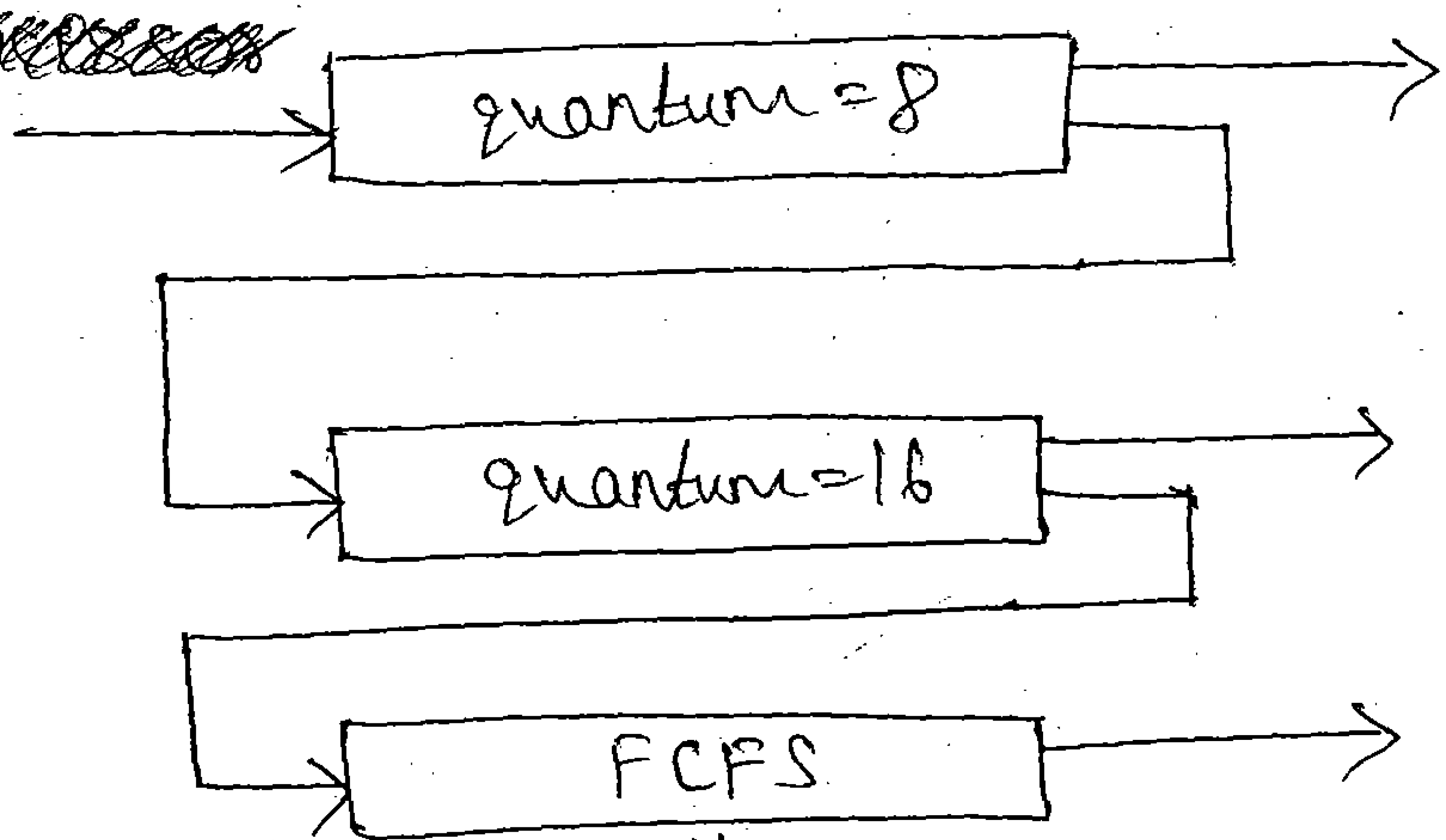
→ A process can move between the various queues; aging can be implemented this way.

→ Multilevel-feedback-queue scheduler defined by the following parameters:
(i) Number of queues
(ii) scheduling algorithms for each queue
(iii) method used to determine when to upgrade a process.
(iv) method used to determine when to demote a process
(v) method used to determine which queue a process will enter when that process needs service.

→ Example:
* Three queues:
→ Q_0 – RR with time quantum 8 milliseconds.

> Q_1 - RR time quantum 16 milliseconds.
> Q_2 - FCFS

* Scheduling
  > A new job enters queue Q_0 which is served FCFS.
    - when it gains CPU, job receives 8 milliseconds.
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1.
  > At Q_1 job is again served FCFS & receives 16 additional milliseconds.
    - If it still does not complete, it is preempted and moved to queue Q_2.



multilevel feedback queues

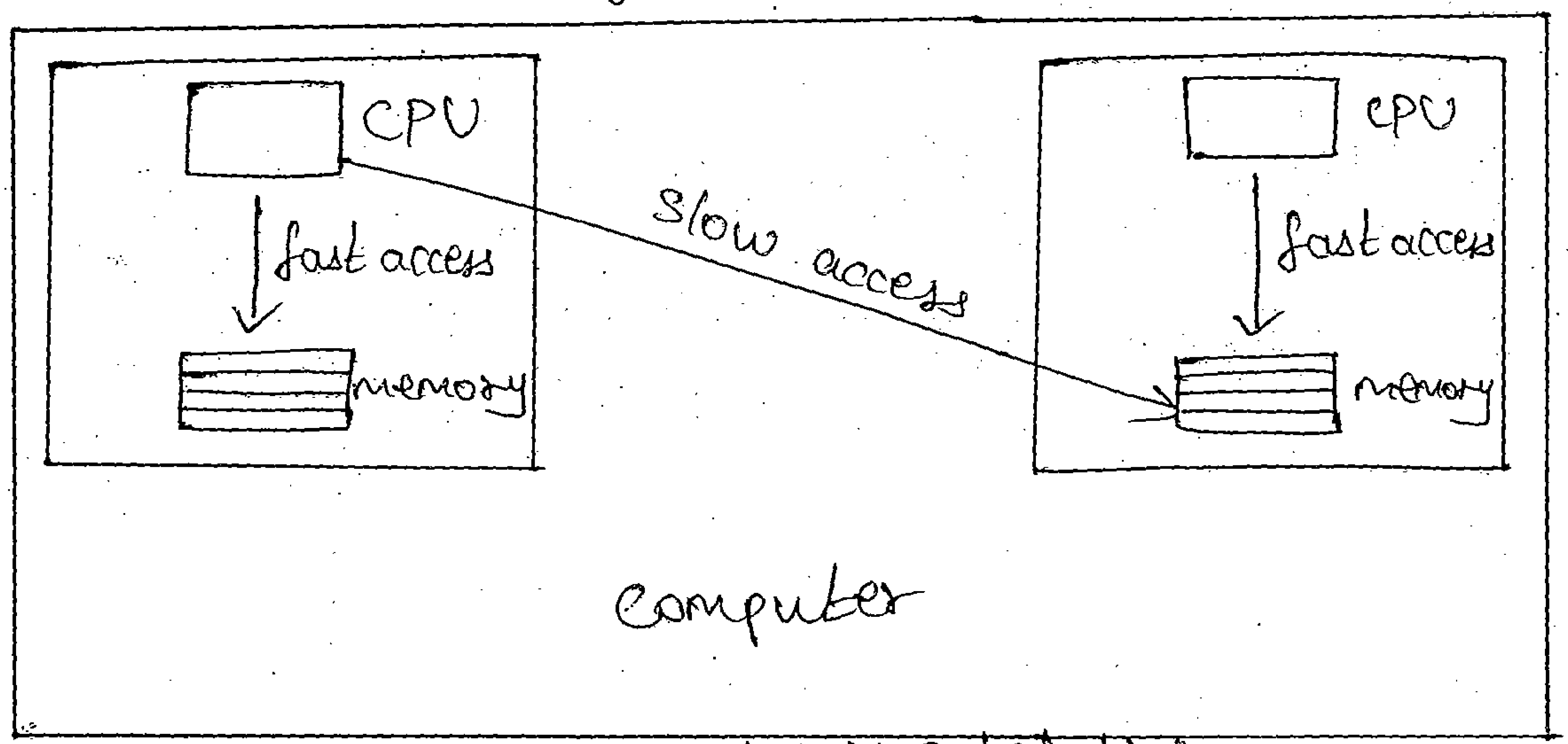
7) Multi-processor scheduling
> CPU scheduling more complex when multiple CPUs are available
> Homogeneous processors - within a multiprocessor.
> Asymmetric multiprocessing - only one processor accesses the system data structures, reducing the need for data sharing.
> Symmetric multiprocessing (SMP) - each processor is self-scheduling, all processes in

-private queue of ready processes.

* Currently, most common.

→ Processor affinity - process has affinity for processor which it is currently running.

* Soft Affinity
* Hard Affinity
* Variations including processor sets.



NUMA and CPU scheduling

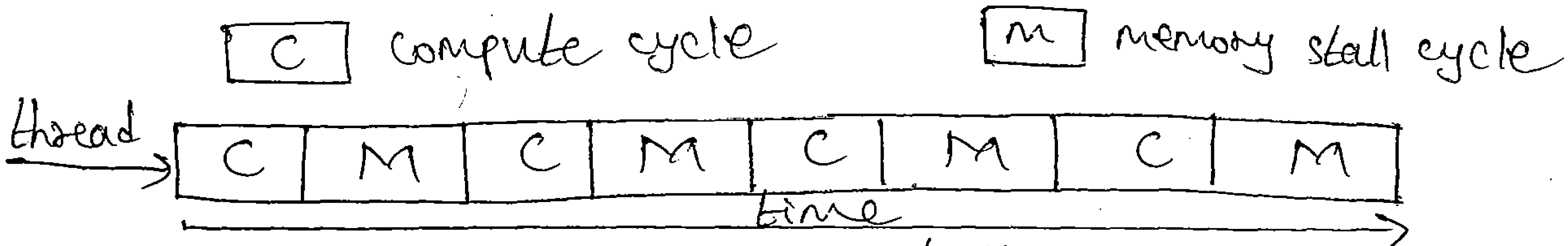
→ Load Balancing - attempts to keep the workload evenly distributed across all processors in an SMP system.

→ Approaches - push migration and pull migration.

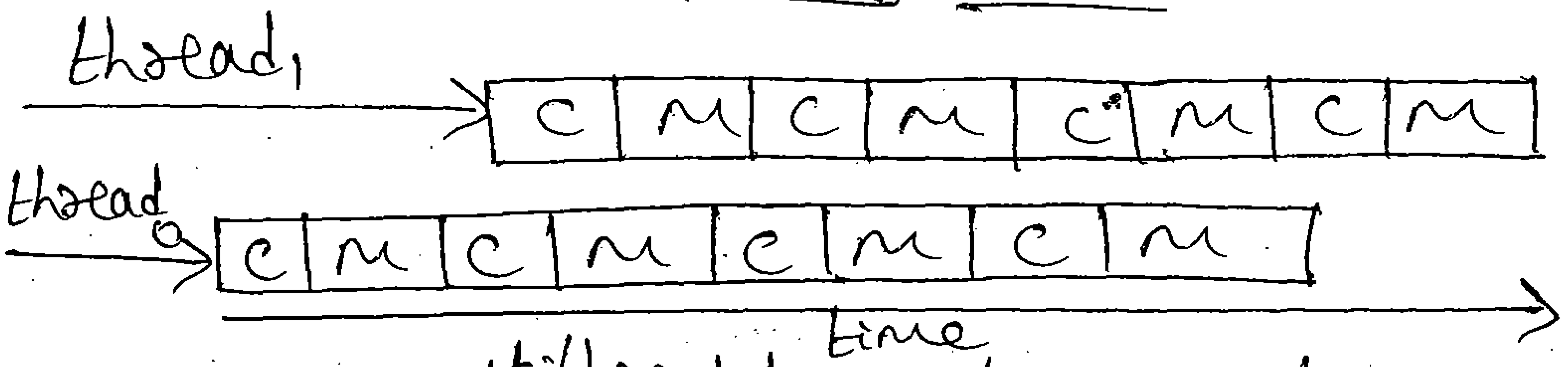
→ Multicore Processors.

* Recent trend to place multiple processor cores on same physical chip.
* Faster and consumes less power.
* Multiple threads per core also growing
  » Takes advantage of memory stall to make progress on another thread while memory retrieve happens.

* Two ways to multithread a processor: Coarse-grained and



memory stall



multithreaded multicore system

→ Virtualization and Scheduling

* Virtualization software schedules multiple guests onto CPU(s).
* Each guest doing its own scheduling
  » Not knowing it doesn't own the CPUs
  » Can result in poor response time.
  » Can effect time-of-day clocks in guests.
* Can undo good scheduling algorithm efforts of guests.

(VII) Thread Scheduling

→ Distinguishing between user-level and kernel-level threads.

→ On OS that support them, it is kernel-level threads.

→ Not processes - that are being scheduled by the OS.

→ Many-to-one and many-to-many models, thread library schedules user-level threads to run on lightweight process (LWP).

* known as process-contention scope (PCS) since scheduling competition is within the process.
* Typically done via priority set by programmer.

→ kernel thread scheduled onto available CPU is system-contention-scope (SCS) — competition among all threads in system.

→ Pthread Scheduling

* API allows specifying either PCS (or) SCS during thread creation.
  - ≫ PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling.
  - ≫ PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.
* The pthread IPC provides two functions for getting and setting the contention scope policy:
  - ≫ pthread_attr_setscope(pthread_attr_t *attr, int scope)
  - ≫ pthread_attr_getscope(pthread_attr_t *attr, int *scope).