ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

OPERATING SYSTEMS

MASTER IN COMPUTER SCIENCE & BUSINESS TECHNOLOGY

IPC & Memory & Disk

Professor: Olivier Perard

Email: operard@faculty.ie.edu

Github:
https://github.com/operard/opsys_parallel/blob/master/mcsbt/README.md

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

The critical-Section Problem

The Critical-Section Problem

- *n* processes competing to use some shared data.
- No assumptions may be made about speeds or the number of CPUs.
- Each process has a code segment, called *Critical Section (CS)*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its CS, no other process is allowed to execute in its CS.

CS Problem Dynamics (1)

- When a process executes code that manipulates shared data (or resource), we say that the process is in it's Critical Section (for that shared data).
- The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple processors).
- So each process must first request permission to enter its critical section.

CS Problem Dynamics (2)

- The section of code implementing this request is called the Entry Section (ES).
- The critical section (CS) might be followed by a Leave/Exit Section (LS).
- The remaining code is the Remainder Section (RS).
- The critical section problem is to design a protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors).

General structure of process P_i (other is P_j)

do {

entry section

critical section

leave section

remainder section

} **while (TRUE);**

- Processes may share some common variables to synchronize their actions.

Solution to Critical-Section Problem

- There are 3 requirements that must stand for a correct solution:
  1. **Mutual Exclusion**
  2. **Progress**
  3. **Bounded Waiting**
- We can check on all three requirements in each proposed solution, even though the non-existence of each one of them is enough for an incorrect solution.

Solution to CS Problem – Mutual Exclusion

1. **Mutual Exclusion** – If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

- Implications:
  - Critical sections better be focused and short.
  - Better not get into an infinite loop in there.
  - If a process somehow halts/waits in its critical section, it must not interfere with other processes.

Solution to CS Problem – Progress

2. **Progress –** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely:
   - If only one process wants to enter, it should be able to.
   - If two or more want to enter, one of them should succeed.

Solution to CS Problem – Bounded Waiting

3. **Bounded Waiting** – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed.
   - No assumption concerning relative speed of the *n* processes.

Types of solutions to CS problem

- Software solutions –
  - algorithms who's correctness does not rely on any other assumptions.
- Hardware solutions –
  - rely on some special machine instructions.
- Operating System solutions –
  - provide some functions and data structures to the programmer through system/library calls.
- Programming Language solutions –
  - Linguistic constructs provided as part of a language.

What about process failures?

- If all 3 criteria (ME, progress, bounded waiting) are satisfied, then a valid solution will provide robustness against failure of a process in its remainder section (RS).
  - since failure in RS is just like having an infinitely long RS.
- However, no valid solution can provide robustness against a process failing in its critical section (CS).
  - A process Pi that fails in its CS does not signal that fact to other processes: for them Pi is still in its CS.

Drawbacks of software solutions

- Software solutions are very delicate ☺.
- Processes that are requesting to enter their critical section are busy waiting (consuming processor time needlessly).
  - If critical sections are long, it would be more efficient to block processes that are waiting.

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Synchronization Solutions

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of locking:
  - Protecting critical regions via locks.
- Uniprocessors – could disable interrupts:
  - Currently running code would execute without preemption.
  - Generally too inefficient on multiprocessor systems:
    - Operating systems using this are not broadly scalable.
- Modern machines provide special atomic (non-interruptible) hardware instructions:
  - Either test memory word and set value at once.
  - Or swap contents of two memory words.

Interrupt Disabling

- On a Uniprocessor:
  - mutual exclusion is preserved but efficiency of execution is degraded: while in CS, we cannot interleave execution with other processes that are in RS.
- On a Multiprocessor:
  - mutual exclusion is not preserved:
    - CS is now atomic but not mutually exclusive (interrupts are not disabled on other processors).

```
Process Pi:
repeat
   disable interrupts
      critical section
   enable interrupts
      remainder section
forever
```

Special Machine Instructions

- Normally, access to a memory location excludes other access to that same location.
- Extension: designers have proposed machines instructions that perform 2 actions atomically (indivisible) on the same memory location (e.g., reading and writing).
- The execution of such an instruction is also mutually exclusive (even on Multiprocessors).

Solution to Critical Section Problem using Locks

- The general layout is of lock solution is:

```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (TRUE);
```

Disadvantages of Special Machine Instructions

- Busy-waiting is employed, thus while a process is waiting for access to a critical section it continues to consume processor time.
- No Progress (Starvation) is possible when a process leaves CS and more than one process is waiting.
- They can be used to provide mutual exclusion but need to be complemented by other mechanisms to satisfy the bounded waiting requirement of the CS problem.
- See next slide for an example.

Mutex Locks

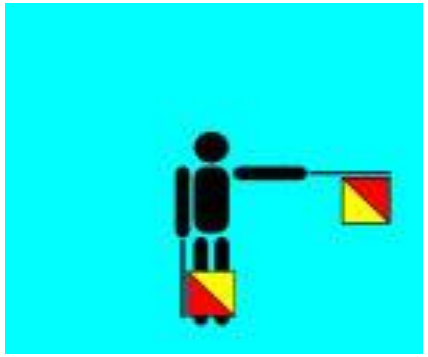
- Previous solutions are complicated and generally inaccessible to application programmers.
- OS designers build software tools to solve critical section problem.
- Simplest is mutex lock.
- Protect a critical section by first `acquire()` a lock then `release()` the lock:
  - Boolean variable indicating if lock is available or not.
- Calls to `acquire()` and `release()` must be atomic:
  - Usually implemented via hardware atomic instructions.
- But this solution requires busy waiting:
  - This lock therefore called a spinlock.

acquire() and release()

```
• acquire() {
        while (!available)
            ; /* busy wait */
        available = false;;
    }
•   release() {
        available = true;
    }
•   do {
    acquire lock
        critical section
    release lock
      remainder section
 } while (true);
```

Semaphores (1)

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Logically, a semaphore S is an integer variable that, apart from initialization, can only be changed through 2 atomic and mutually exclusive operations:
  - wait(S) (also down(S), P(S))
  - signal(S) (also up(S), V(S)
- Less complicated.



Critical Section of *n* Processes

- Shared data:

  semaphore mutex; // initiallized *to* 1

- Process *Pi:*

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (TRUE);
```

Semaphores (2)

- Access is via two atomic operations:

wait (*S*):

```
while (S <= 0);
S--;
```

signal (*S*):

```
S++;
```

Semaphores (3)

- Must guarantee that no 2 processes can execute wait() and signal() on the same semaphore at the same time.
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Could now have busy waiting in CS implementation:
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied.
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Semaphores (4)

- To avoid busy waiting, when a process has to wait, it will be put in a blocked queue of processes waiting for the same event.
- Hence, in fact, a semaphore is a record (structure):

```
type semaphore = record
                   count: integer;
                   queue: list of process
                 end;
var S: semaphore;
```

- With each semaphore there is an associated waiting queue.
- Each entry in a waiting queue has 2 data items:
  - value (of type integer)
  - pointer to next record in the list

Semaphore's operations

- When a process must wait for a semaphore S, it is blocked and put on the semaphore's queue.
- Signal operation removes (assume a fair policy like FIFO) first process from the queue and puts it on list of ready processes.

```
wait(S):
  S.count--;
  if (S.count<0) {
    block this process
    place this process in S.queue
  }

signal(S):
  S.count++;
  if (S.count<=0) {
    remove a process P from S.queue
    place this process P on ready list
  }
```

Semaphore Implementation (1)

- Define semaphore as a C struct:

```
typedef struct {
   int value;
   struct process *list;
} semaphore;
```

- Assume two operations:
  - **Block:** place the process invoking the operation on the appropriate waiting queue.
  - **Wakeup:** remove one of processes in the waiting queue and place it in the ready queue.

Semaphore Implementation (2)

- Semaphore operations now defined as

```
wait (semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
}

signal (semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

Two Types of Semaphores

1. *Counting* semaphore – integer value can range over an unrestricted domain.
2. *Binary* semaphore – integer value can range only between 0 and 1 (really a Boolean); can be simpler to implement (use waitB and signalB operations); same as Mutex lock.

- We can implement a counting semaphore *S* using 2 binary semaphores (that protect its counter).

Semaphore as a General Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore *flag* initialized to 0.
- Code:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of waiting processes.
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue (say, if LIFO) in which it is suspended.
- **Priority Inversion** – scheduling problem when lower-priority process holds a lock needed by higher-priority process

Problems with Semaphores

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes.
- But wait(S) and signal(S) are scattered among several processes. Hence, difficult to understand their effects.
- Usage must be correct in all the processes (correct order, correct variables, no omissions).
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- One bad (or malicious) process can fail the entire collection of processes.

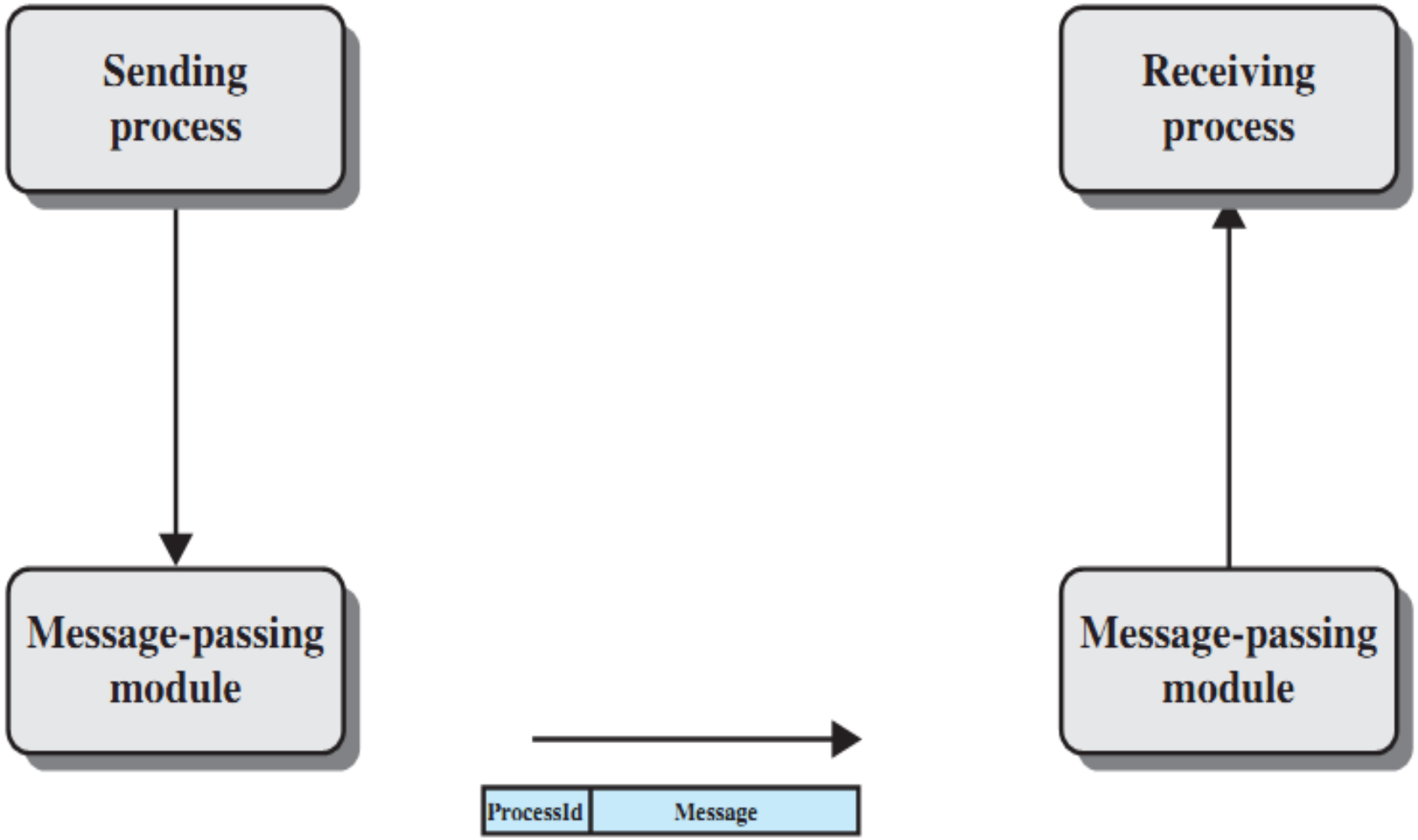
ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Inter-process Communication

Inter-Process Communication (IPC)

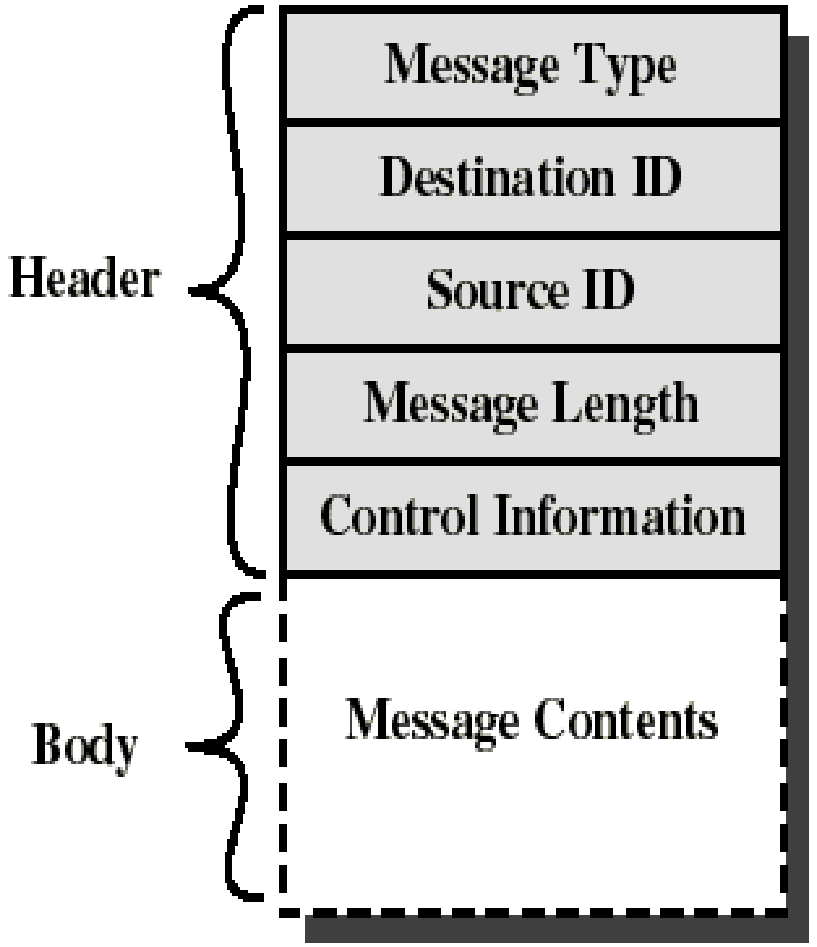
- Mechanism for processes to communicate and to synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
- We have at least two primitives:
  - **send***(destination, message*) or **send**(*message*)
  - **receive***(source, message*) or **receive**(*message*)
- Message size is fixed or variable.

Basic Message-passing Primitives

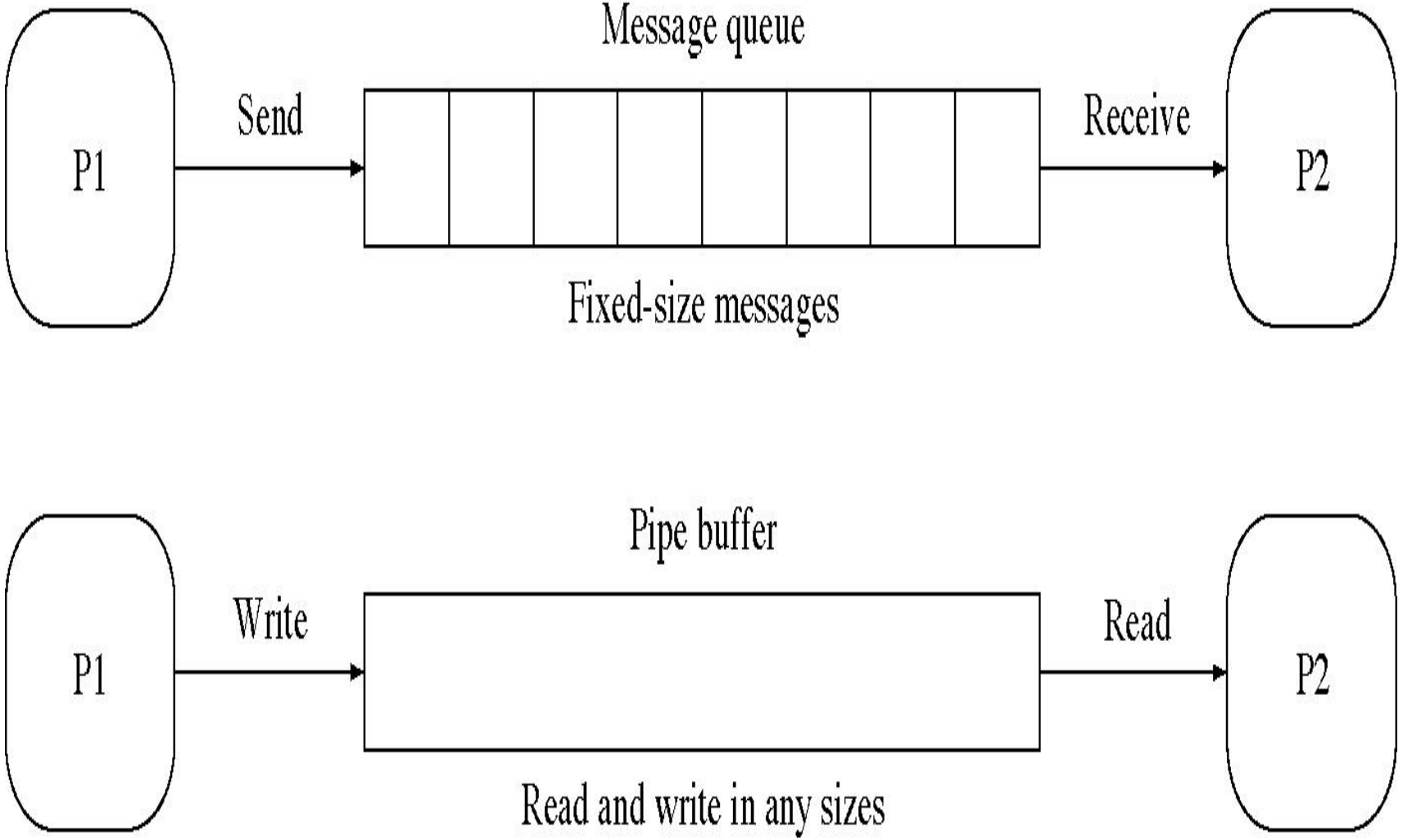


Message format

- Consists of header and body of message.
- In Unix: no ID, only message type.
- Control info:
  - what to do if run out of buffer space.
  - sequence numbers.
  - priority.
- **Queuing discipline: usually FIFO but can also include priorities.**



Messages and Pipes Compared



Message Passing

- Message passing is a general method used for IPC:
  - for processes inside the same computer.
  - for processes in a networked/distributed system.
- In both cases, the process may or may not be blocked while sending a message or attempting to receive a message.

Synchronization in message passing (1)

- Message passing may be blocking or non-blocking.
- **Blocking** is considered **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 **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Synchronization in message passing (2)

- For the sender: it is more natural not to be blocked after issuing send:
  - can send several messages to multiple destinations.
  - but sender usually expect acknowledgment of message receipt (in case receiver fails).
- For the receiver: it is more natural to be blocked after issuing receive:
  - the receiver usually needs the information before proceeding.
  - but could be blocked indefinitely if sender process fails before send.

Synchronization in message passing (3)

- Hence other possibilities are sometimes offered.
- Example: blocking send, blocking receive:
  - both are blocked until the message is received.
  - occurs when the communication link is unbuffered (no message queue).
  - provides tight synchronization (*rendezvous*).

Synchronization in message passing (4)

- There are really 3 combinations here that make sense:

1. Blocking send, Blocking receive
2. Nonblocking send, Nonblocking receive
3. Nonblocking send, Blocking receive – most popular – example:
    - Server process that provides services/resources to other processes. It will need the expected information before proceeding.

IPC Requirements

- If *P* and *Q* wish to communicate, they need to:
  - establish communication link between them.
  - exchange messages via send/receive.
- Implementation of communication link:
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Link Capacity – Buffering

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

1. Zero capacity – 0 messages
   Sender must wait for receiver (rendezvous).
2. Bounded capacity – finite length of *n* messages
   Sender must wait if link full.
3. Unbounded capacity – infinite length
   Sender never waits.

Direct/Indirect Communication

- Direct communication:
  - when a specific process identifier is used for source/destination.
  - but it might be impossible to specify the source ahead of time (e.g., a print server).
- Indirect communication (more convenient):
  - messages are sent to a shared mailbox which consists of a queue of messages.
  - senders place messages in the mailbox, receivers pick them up.

Direct Communication

- Processes must name each other explicitly:
  - **send**(*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from 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.

Indirect Communication (1)

- 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.
- 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 communication links.
  - Link may be unidirectional or bi-directional.

Indirect Communication (2)

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:

send(*A, message*) – send a message to mailbox A.

receive(*A, message*) – receive a message from mailbox A.

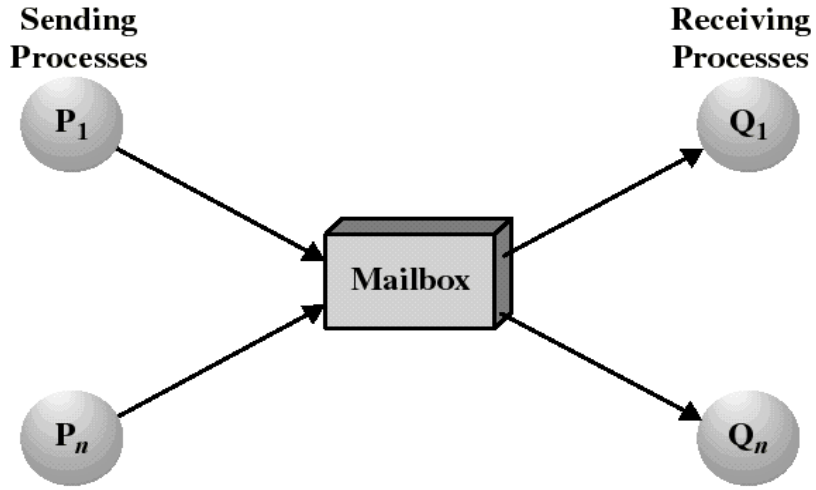
Indirect Communication (3)

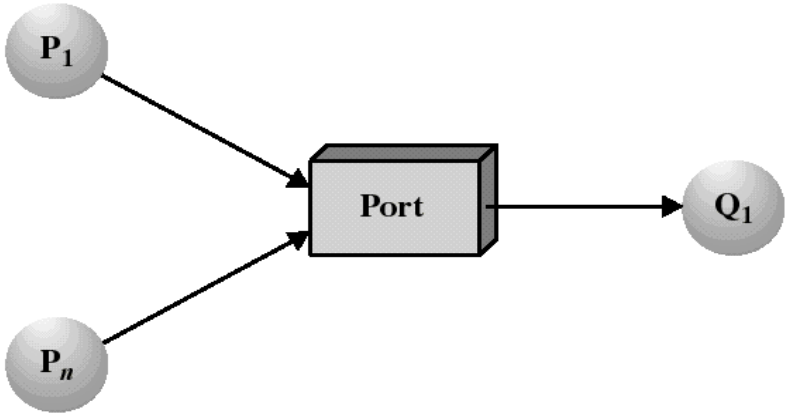
- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A.
  - P_1, sends; P_2 and P_3 receive.
  - Who gets the message?

- Possible 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.

Mailboxes and Ports

- A mailbox can be private to one sender/receiver pair.
- The same mailbox can be shared among several senders and receivers:
  - the OS may then allow the use of message types (for selection).
- Port: is a mailbox associated with one receiver and multiple senders
  - used for client/server applications: the receiver is the server.





Ownership of ports and mailboxes

- A port is usually own and created by the receiving process.
- The port is destroyed when the receiver terminates.
- The OS creates a mailbox on behalf of a process (which becomes the owner).
- The mailbox is destroyed at the owner's request or when the owner terminates.

Mutual Exclusion – Message Passing

- create a mailbox ***mutex*** shared by n processes.
- send() is non-blocking.
- receive() blocks when ***mutex*** is empty.
- Initialization: send(***mutex***, “go”);
- The first Pi who executes receive() will enter CS. Others will be blocked until Pi resends msg.

```
Process Pi:
var msg: message;
repeat
   receive(mutex,msg);
   CS
   send(mutex,msg);
   RS
forever
```

Bounded-Buffer – Message Passing

- The producer place items (inside messages) in the mailbox ***mayconsume***.
- ***mayconsume*** acts as our buffer: consumer can consume item when at least one message present.
- Mailbox ***mayproduce*** is filled initially with k null messages (k= buffer size).
- The size of ***mayproduce*** shrinks with each production and grows with each consumption.
- Solution can support multiple producers/consumers.

Bounded-Buffer – Message Passing

```
Producer:
var pmsg: message;
repeat
  receive(mayproduce, pmsg);
  pmsg := produce();
  send(mayconsume, pmsg);
forever

Consumer:
var cmsg: message;
repeat
  receive(mayconsume, cmsg);
  consume(cmsg);
  send(mayproduce, null);
forever
```

P/C Problem with Message Passing (1)

```
#define N 100                                   /* number of slots in the buffer */

void producer(void)
{
     int item;
     message m;                                 /* message buffer */

     while (TRUE) {
          item = produce_item( );               /* generate something to put in buffer */
          receive(consumer, &m);                /* wait for an empty to arrive */
          build_message(&m, item);              /* construct a message to send */
          send(consumer, &m);                   /* send item to consumer */
     }
}
```

P/C Problem with Message Passing (2)

```
void consumer(void)
{
        int item, i;
        message m;

        for (i = 0; i < N; i++) send(producer, &m);  /* send N empties */
        while (TRUE) {
                receive(producer, &m);              /* get message containing item */
                item = extract_item(&m);            /* extract item from message */
                send(producer, &m);                 /* send back empty reply */
                consume_item(item);                 /* do something with the item */
        }
}
```

Examples of IPC Systems – POSIX

- POSIX Shared Memory example
- Process first creates shared memory segment

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

- Process wanting access to that shared memory must attach to it

```
shared_memory = (char *) shmat(segment_id, NULL, 0);
```

- Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```

- When done a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

- Now process can remove the shared memory segment

```
shmdt(shared_id, IPC_RMID, NULL);
```

Examples of IPC Systems – Mach

- Mach communication is message based:
  - Even system calls are messages.
  - Each task gets two mailboxes at creation: Kernel and Notify.
  - Only three system calls needed for message transfer:

`msg_send(), msg_receive(), msg_rpc()`

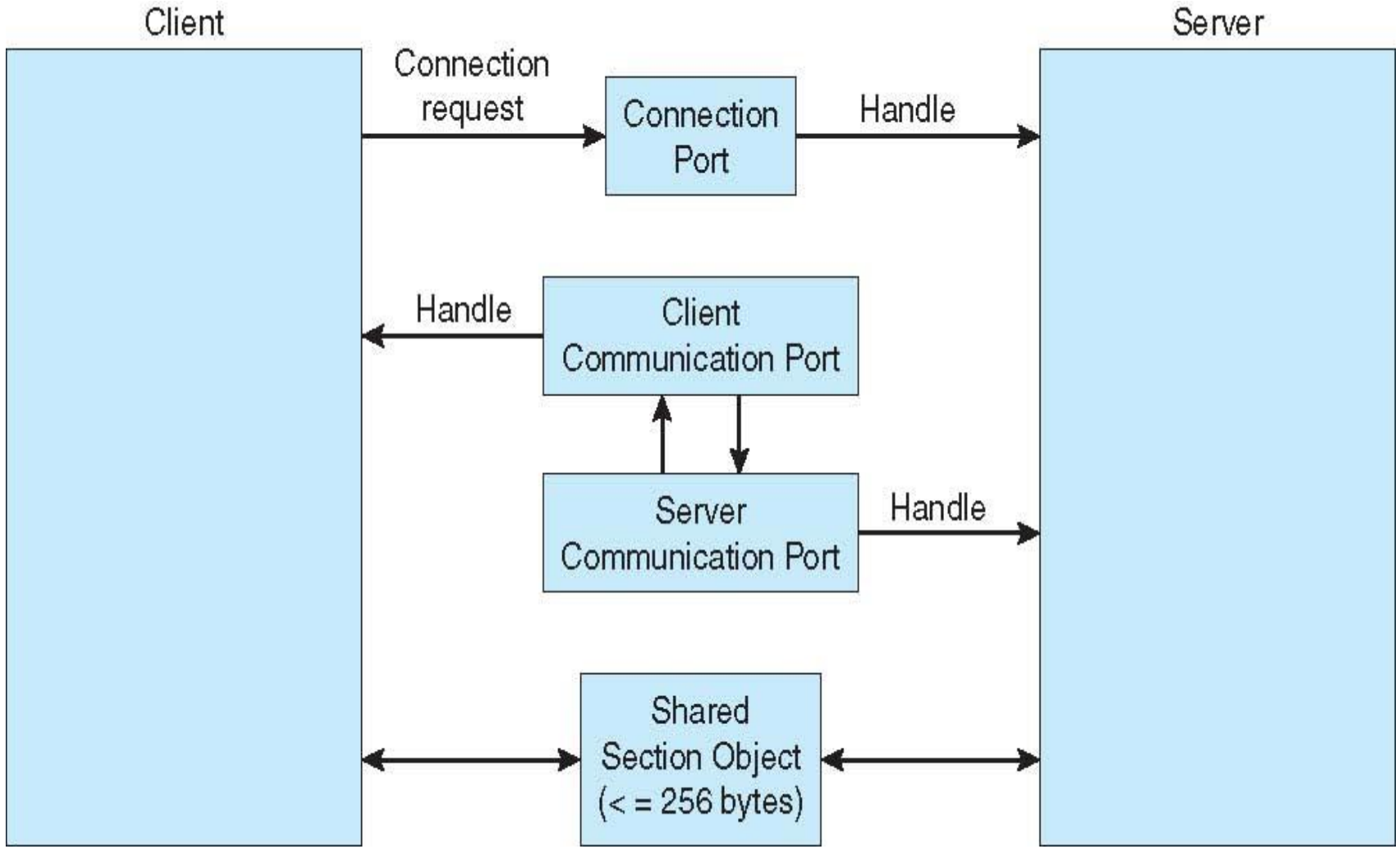
  - Mailboxes needed for communication, created via

`port_allocate()`

Examples of IPC Systems – Windows XP

- Message-passing centric via LPC facility:
  - Only works between processes on the same system.
  - Uses ports (like mailboxes) to establish and maintain communication channels.
  - Communication works as follows:
    - The client opens a handle to the subsystem's connection port object.
    - The client sends a connection request.
    - The server creates two private communication ports and returns the handle to one of them to the client.
    - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

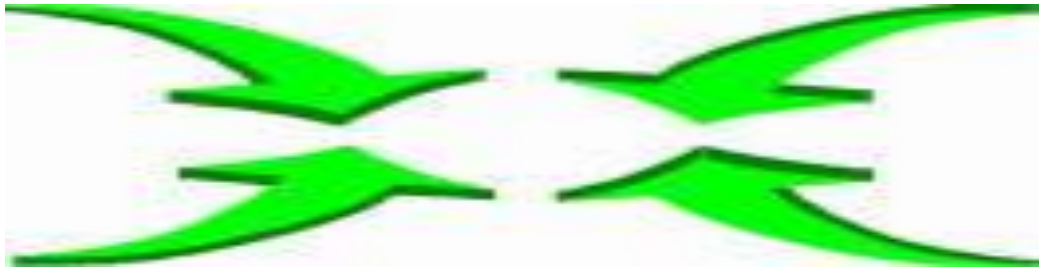
Local Procedure Calls in Windows XP



Communications in Client-Server Systems

- There are various mechanisms:

1. Pipes
2. Sockets (Internet)
3. Remote Procedure Calls (RPCs)
4. Remote Method Invocation (RMI, Java)



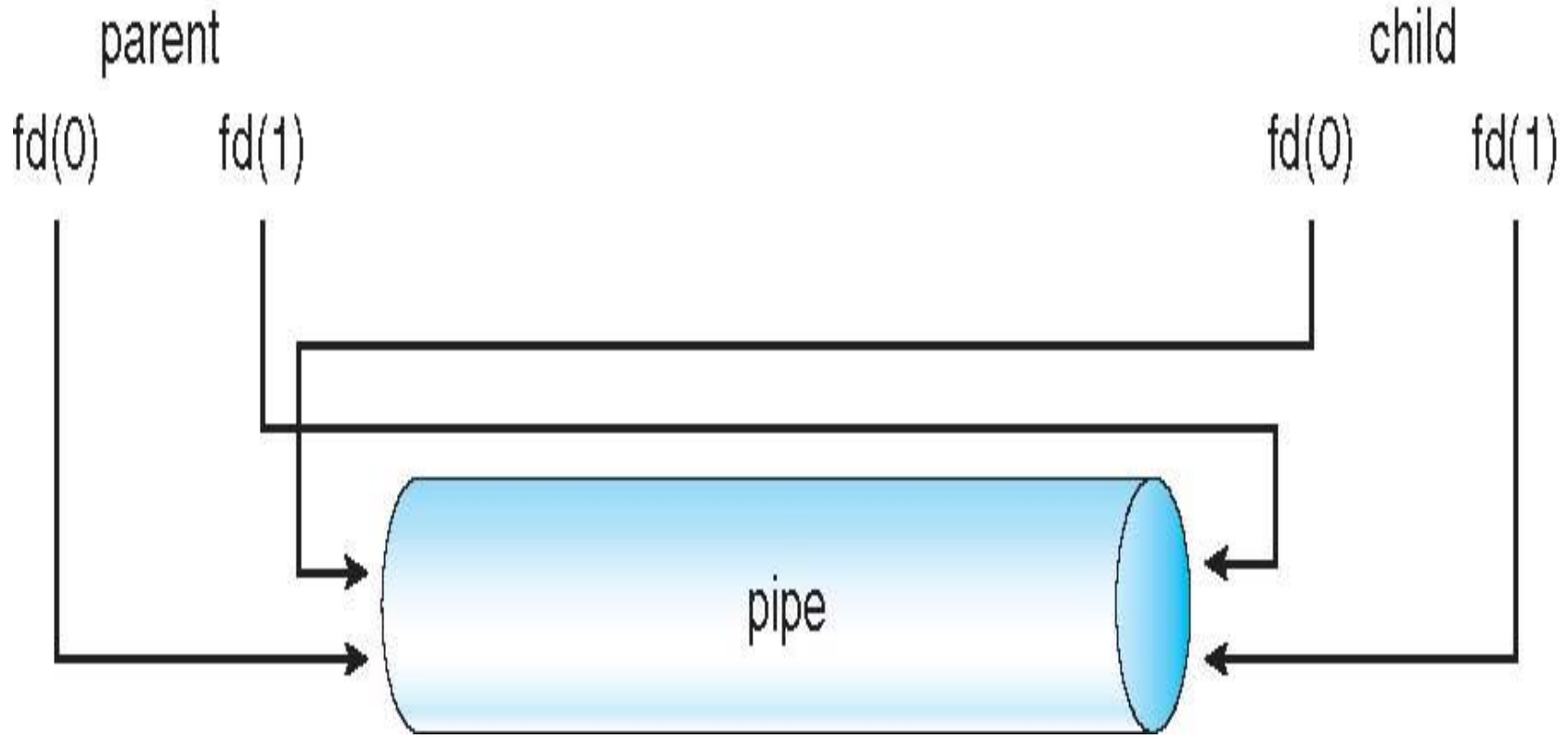
Pipes

- Acts as a conduit allowing two processes to communicate.
- Some issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., parent-child) between the communicating processes?
  - Can the pipes be used over a network?

Ordinary Pipes

- **Ordinary Pipes** allow communication in standard producer-consumer style.
- Producer writes to one end (the *write-end* of the pipe).
- Consumer reads from the other end (the *read-end* of the pipe).
- Ordinary pipes are therefore unidirectional.
- Require parent-child relationship between communicating processes.

Ordinary Pipes



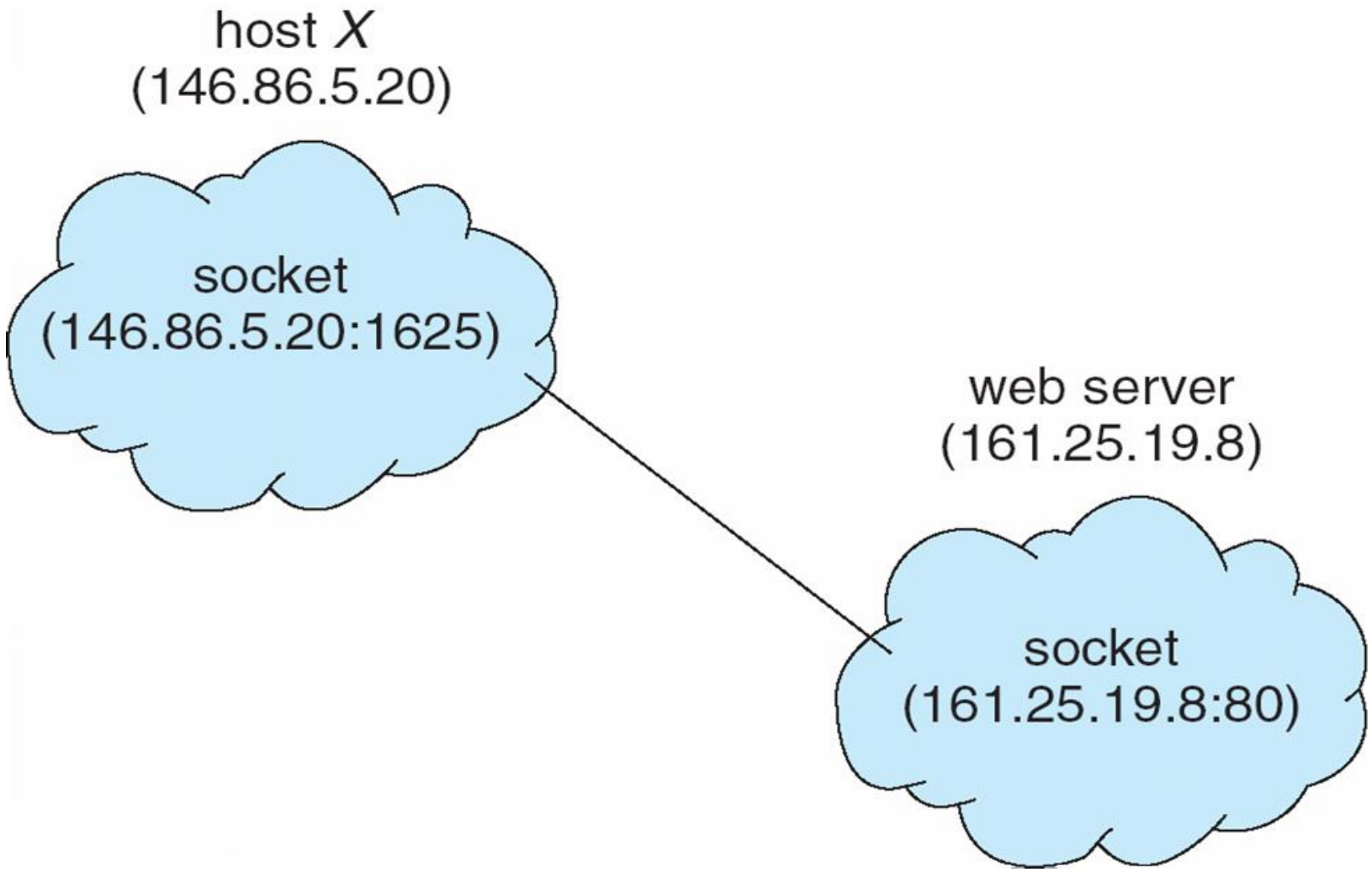
Named Pipes

- Named Pipes are more powerful than ordinary pipes.
- Communication is bidirectional.
- No parent-child relationship is necessary between the communicating processes.
- Several processes can use the named pipe for communication.
- Provided on both UNIX and Windows systems.

Sockets

- A socket is defined as an *endpoint for communication.*
- Concatenation of IP address and port.
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8.**
- Communication consists between a pair of sockets.

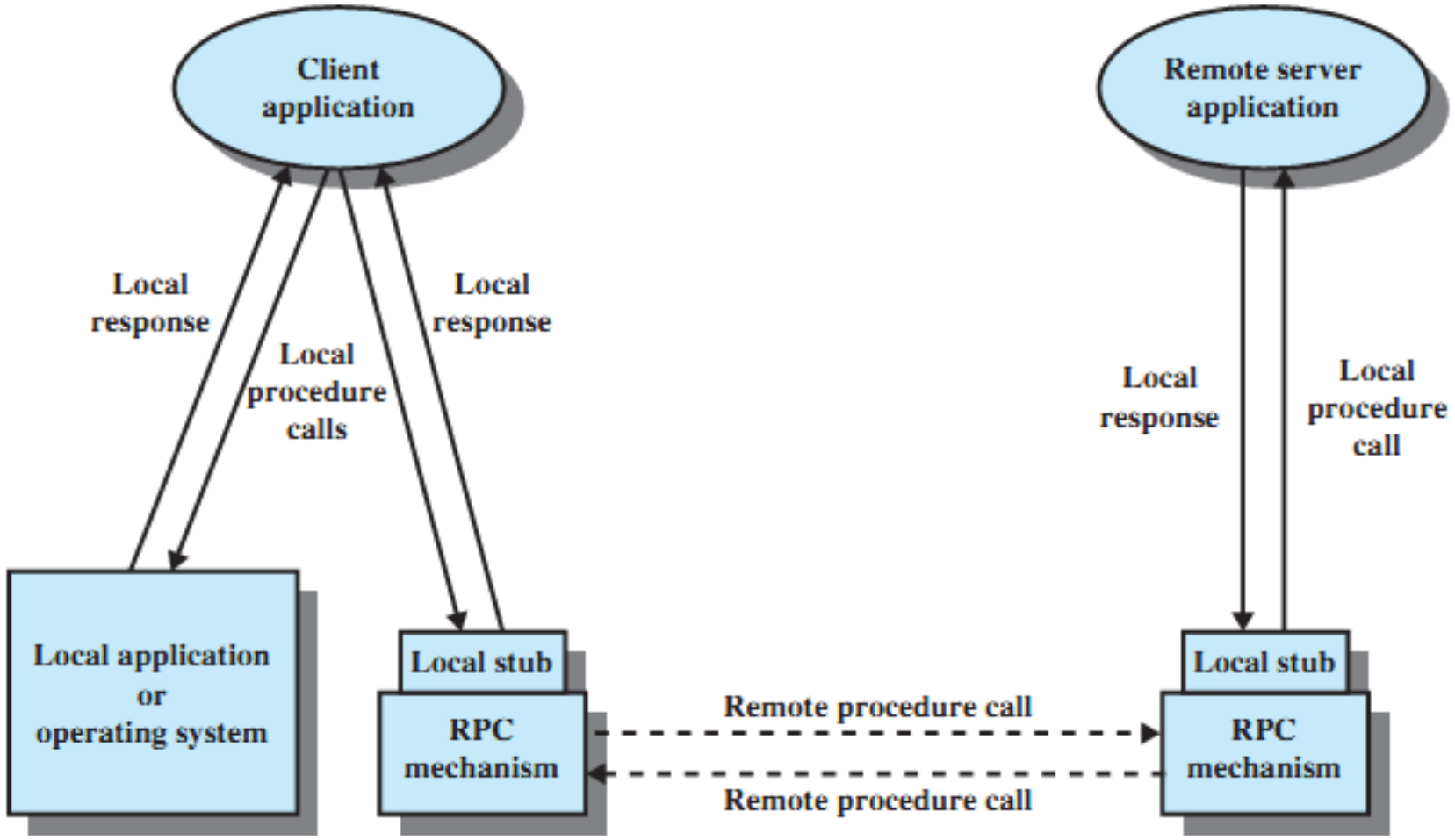
Socket Communication



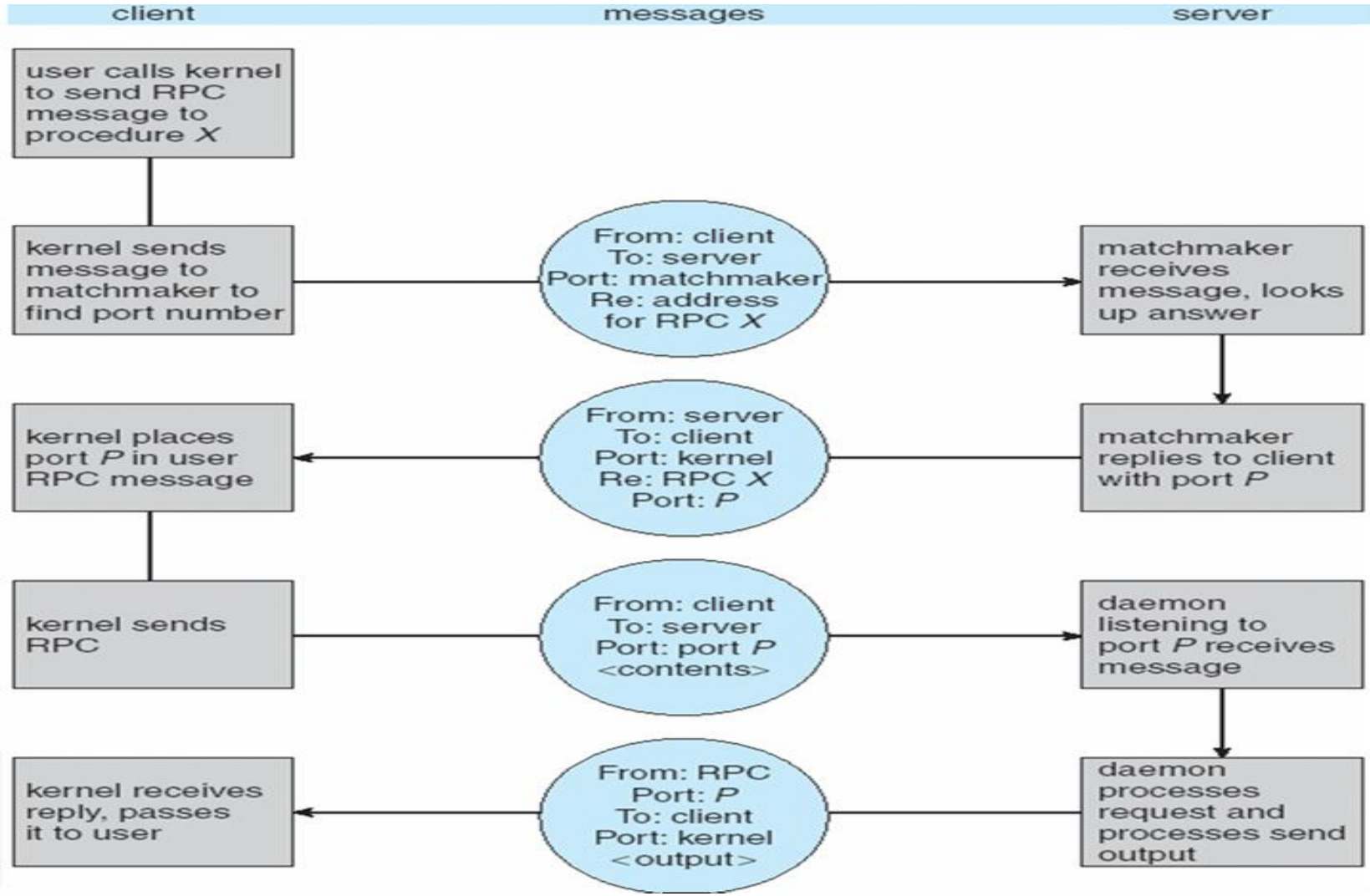
Remote Procedure Calls (RPCs)

- RPC abstracts a Local Procedure Call (LPC) between processes on a networked system.
- **Stubs** – client-side proxy for the actual procedure existing on the server.
- The client-side stub locates the server and *marshals* the parameters.
- The server-side stub/skeleton receives this message, unpacks the marshaled parameters, and performs the procedure on the server.
- Vice versa happens on the opposite direction.

Remote Procedure Call Mechanism

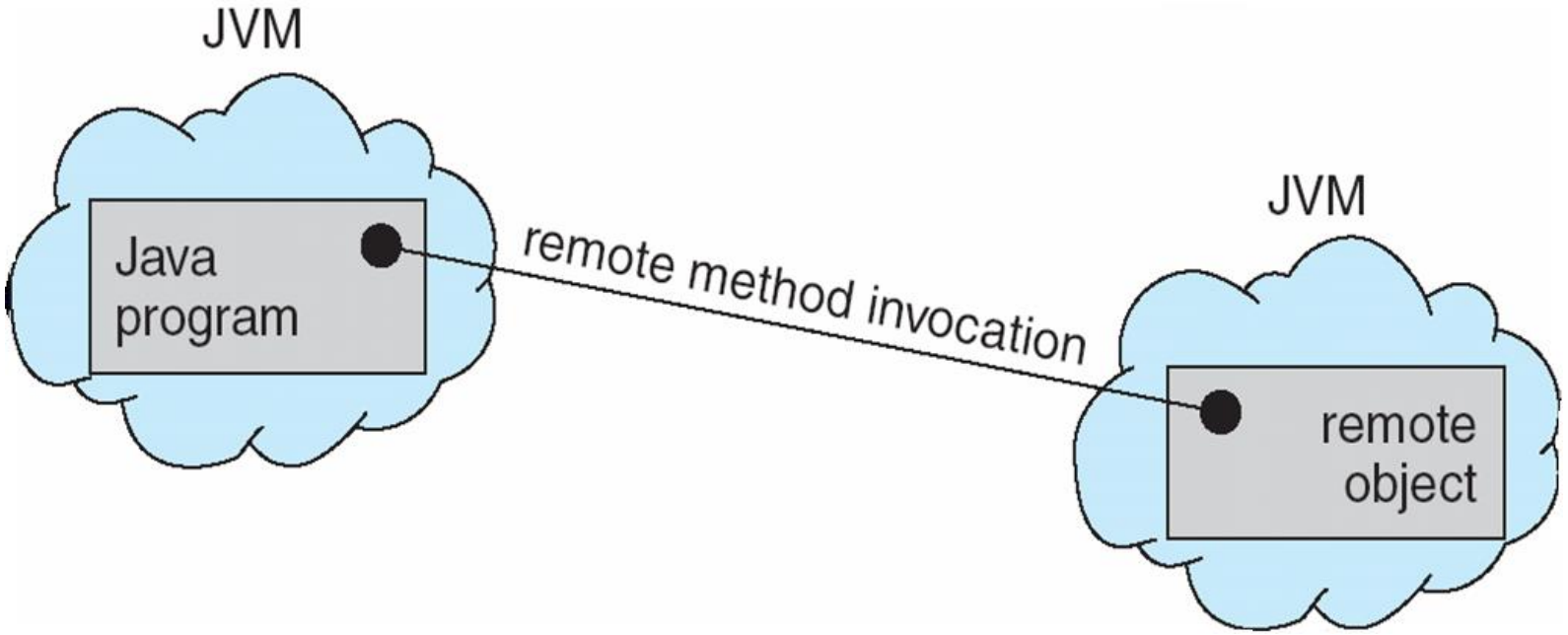


Execution of RPC

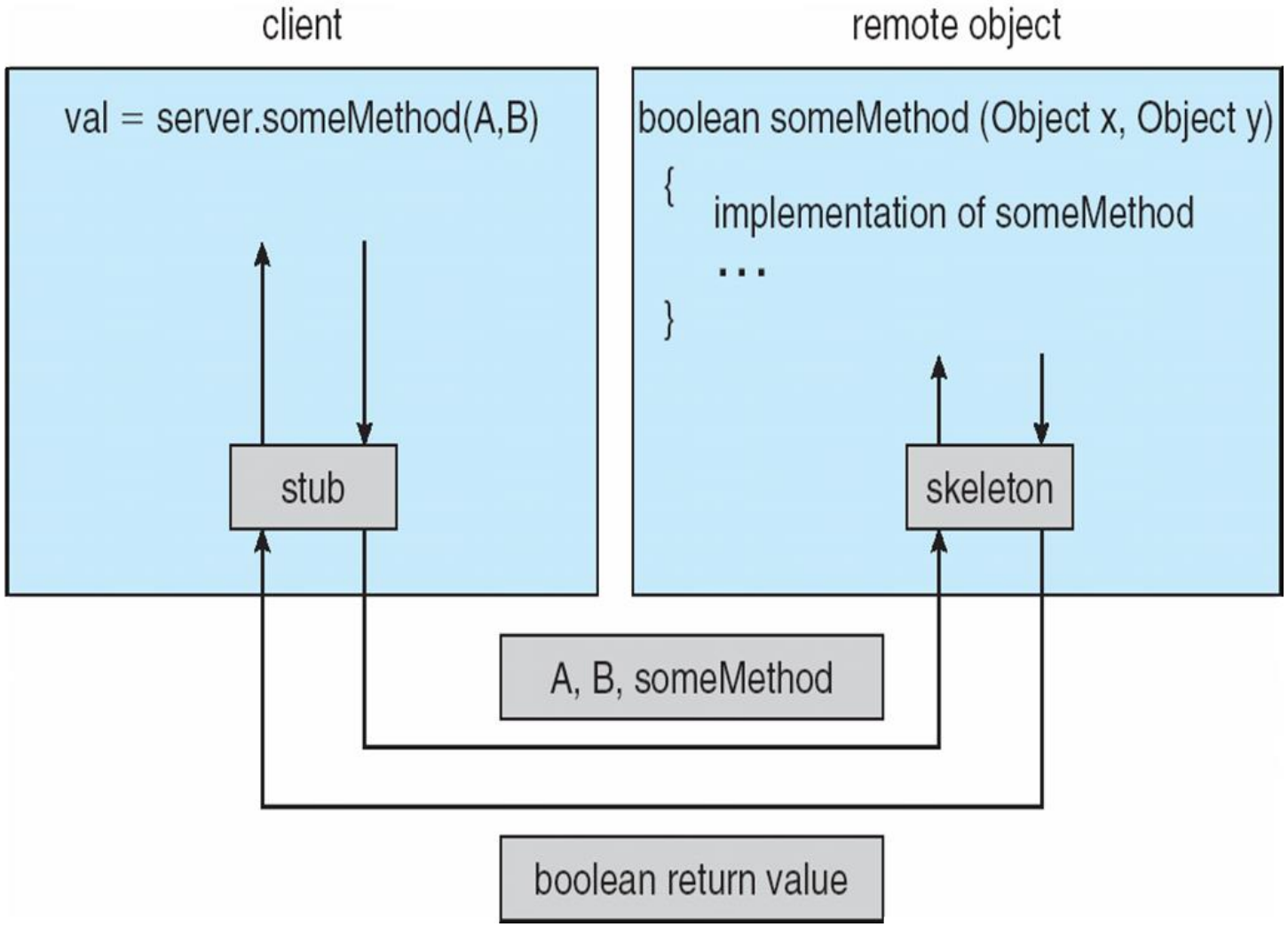


Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.



(Un)Marshalling Parameters



Python Implementation

- https://docs.python.org/3.4/library/ipc.html
- https://docs.python.org/3/library/ipc.html

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Passing Messages to Processes (Queue)

```
import multiprocessing

class MyFancyClass(object):

    def __init__(self, name):
        self.name = name

    def do_something(self):
        proc_name = multiprocessing.current_process().name
        print 'Doing something fancy in %s for %s!' % (proc_name, self.name)


def worker(q):
    obj = q.get()
    obj.do_something()


if __name__ == '__main__':
    queue = multiprocessing.Queue()

    p = multiprocessing.Process(target=worker, args=(queue,))
    p.start()

    queue.put(MyFancyClass('Fancy Dan'))

    # Wait for the worker to finish
    queue.close()
    queue.join_thread()
    p.join()
```

```
$ python multiprocessing_queue.py

Doing something fancy in Process-1 for Fancy Dan!
```

Signaling between Processes

```
import multiprocessing
import time

def wait_for_event(e):
    """Wait for the event to be set before doing anything"""
    print 'wait_for_event: starting'
    e.wait()
    print 'wait_for_event: e.is_set()->', e.is_set()

def wait_for_event_timeout(e, t):
    """Wait t seconds and then timeout"""
    print 'wait_for_event_timeout: starting'
    e.wait(t)
    print 'wait_for_event_timeout: e.is_set()->', e.is_set()


if __name__ == '__main__':
    e = multiprocessing.Event()
    w1 = multiprocessing.Process(name='block',
                                 target=wait_for_event,
                                 args=(e,))
    w1.start()

    w2 = multiprocessing.Process(name='non-block',
                                 target=wait_for_event_timeout,
                                 args=(e, 2))
    w2.start()

    print 'main: waiting before calling Event.set()'
    time.sleep(3)
    e.set()
    print 'main: event is set'
```

```
$ python -u multiprocessing_event.py

main: waiting before calling Event.set()
wait_for_event: starting
wait_for_event_timeout: starting
wait_for_event_timeout: e.is_set()-> False
main: event is set
wait_for_event: e.is_set()-> True
```

Synchronizing Processes

```
import multiprocessing
import time

def stage_1(cond):
    """perform first stage of work, then notify stage_2 to continue"""
    name = multiprocessing.current_process().name
    print 'Starting', name
    with cond:
        print '%s done and ready for stage 2' % name
        cond.notify_all()

def stage_2(cond):
    """wait for the condition telling us stage_1 is done"""
    name = multiprocessing.current_process().name
    print 'Starting', name
    with cond:
        cond.wait()
        print '%s running' % name

if __name__ == '__main__':
    condition = multiprocessing.Condition()
    s1 = multiprocessing.Process(name='s1', target=stage_1, args=(condition,))
    s2_clients = [
        multiprocessing.Process(name='stage_2[%d]' % i, target=stage_2, args=(condition,))
        for i in range(1, 3)
        ]

    for c in s2_clients:
        c.start()
        time.sleep(1)
    s1.start()

    s1.join()
    for c in s2_clients:
        c.join()
```

```
$ python multiprocessing_condition.py

Starting s1
s1 done and ready for stage 2
Starting stage_2[1]
stage_2[1] running
Starting stage_2[2]
stage_2[2] running
```

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Processes Pool

```
import multiprocessing

def do_calculation(data):
    return data * 2

def start_process():
    print 'Starting', multiprocessing.current_process().name

if __name__ == '__main__':
    inputs = list(range(10))
    print 'Input   :', inputs

    builtin_outputs = map(do_calculation, inputs)
    print 'Built-in:', builtin_outputs

    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(processes=pool_size,
                                initializer=start_process,
                                )
    pool_outputs = pool.map(do_calculation, inputs)
    pool.close() # no more tasks
    pool.join()  # wrap up current tasks

    print 'Pool    :', pool_outputs
```

```
$ python multiprocessing_pool.py

Input   : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Built-in: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
Starting PoolWorker-11
Starting PoolWorker-12
Starting PoolWorker-13
Starting PoolWorker-14
Starting PoolWorker-15
Starting PoolWorker-16
Starting PoolWorker-1
Starting PoolWorker-2
Starting PoolWorker-3
Starting PoolWorker-4
Starting PoolWorker-5
Starting PoolWorker-8
Starting PoolWorker-9
Starting PoolWorker-6
Starting PoolWorker-10
Starting PoolWorker-7
Pool    : [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

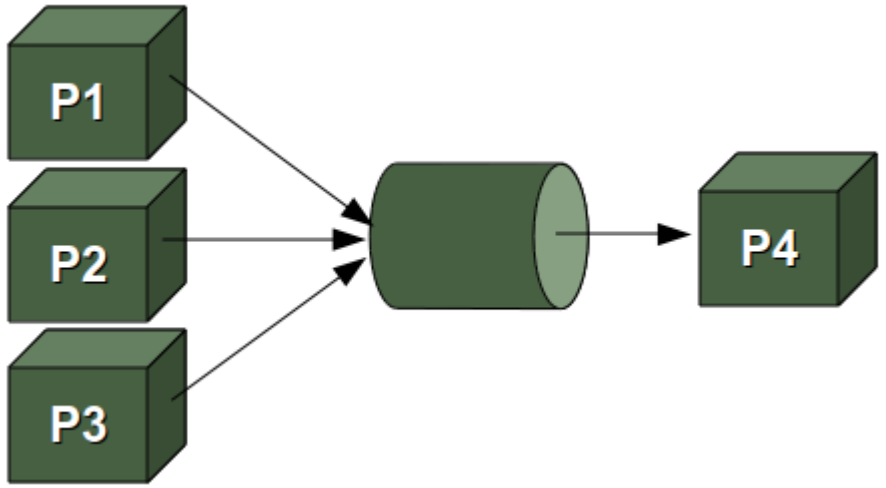
Client/Server with Sockets

```
from multiprocessing.connection import Client

address = ('localhost', 6000)
conn = Client(address, authkey='secret password')
conn.send('close')
# can also send arbitrary objects:
# conn.send(['a', 2.5, None, int, sum])
conn.close()
```

```
from multiprocessing.connection import Listener

address = ('localhost', 6000)     # family is deduced to be 'AF_INET'
listener = Listener(address, authkey='secret password')
conn = listener.accept()
print 'connection accepted from', listener.last_accepted
while True:
    msg = conn.recv()
    # do something with msg
    if msg == 'close':
        conn.close()
        break
listener.close()
```

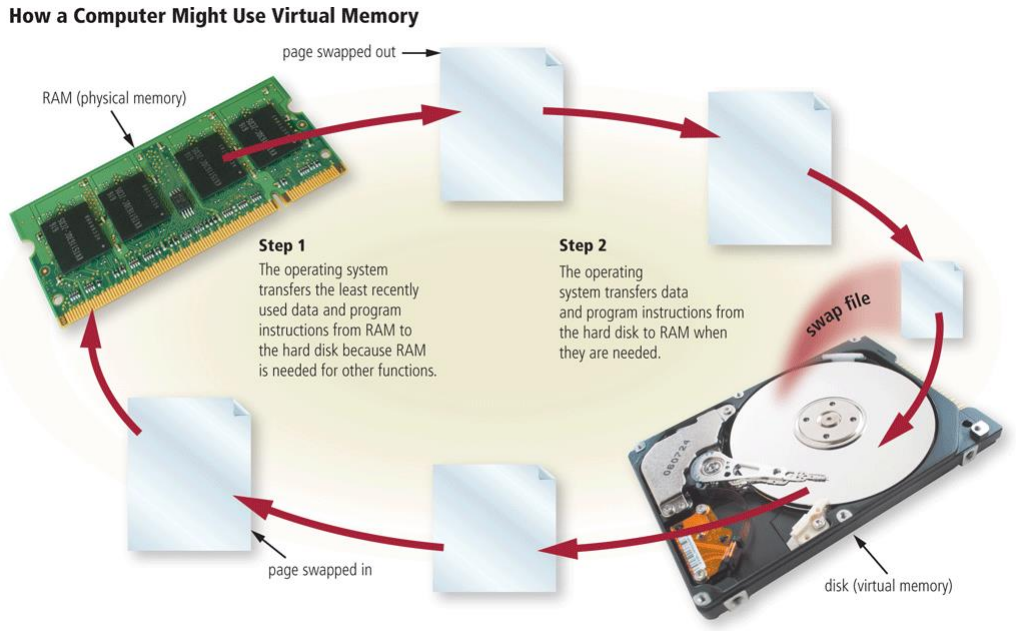
Named Pipe, FIFOs

```
import os, time, sys
pipe_name = 'pipe_test'

def child( ):
    pipeout = os.open(pipe_name, os.O_WRONLY)
    counter = 0
    while True:
        time.sleep(1)
        os.write(pipeout, 'Number %03d\n' % counter)
        counter = (counter+1) % 5

def parent( ):
    pipein = open(pipe_name, 'r')
    while True:
        line = pipein.readline()[:-1]
        print 'Parent %d got "%s" at %s' % (os.getpid(), line, time.time( ))

if not os.path.exists(pipe_name):
    os.mkfifo(pipe_name)
pid = os.fork()
if pid != 0:
    parent()
else:
    child()
```



Operating systems Functions: Memory

- Memory management optimizes the use of the computer or device's internal memory
- **Virtual memory** is a portion of a storage medium functioning as additional RAM



Memory management

- Basic memory management
- Swapping
- Virtual memory
- Page replacement algorithms
- Modeling page replacement algorithms
- Design issues for paging systems
- Implementation issues
- Segmentation

In an ideal world…

- The ideal world has memory that is
  - Very large
  - Very fast
  - Non-volatile (doesn't go away when power is turned off)
- The real world has memory that is:
  - Very large
  - Very fast
  - Affordable!

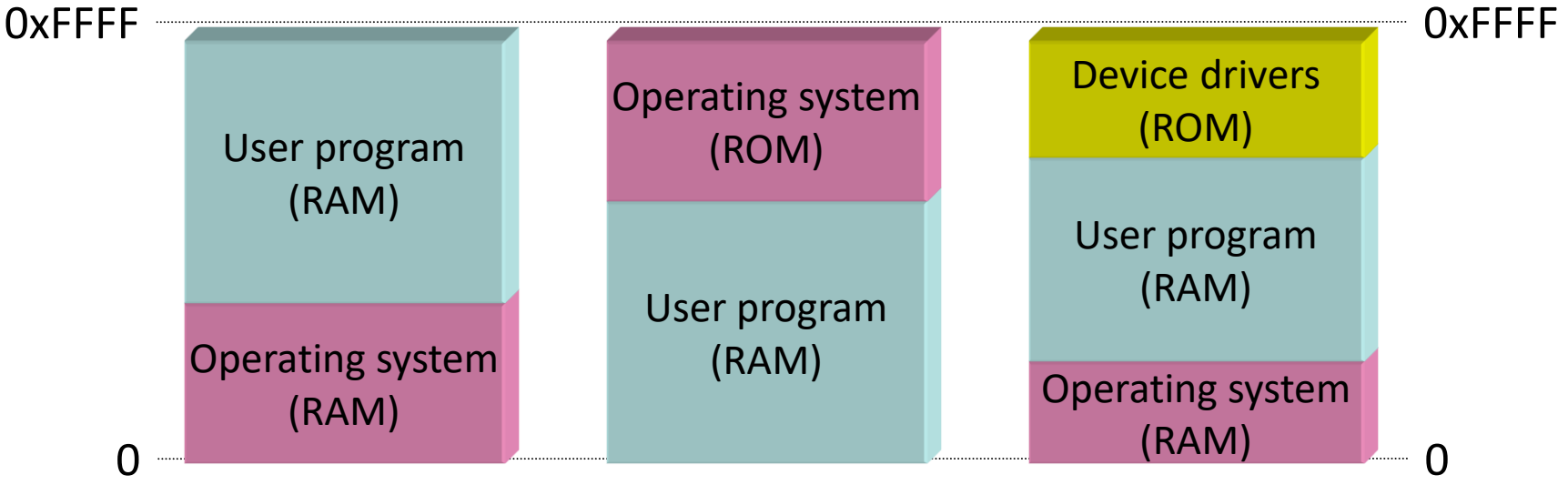
  ⇒Pick any two…
- Memory management goal: make the real world look as much like the ideal world as possible

Memory hierarchy

- What is the memory hierarchy?
  - Different levels of memory
  - Some are small & fast
  - Others are large & slow
- What levels are usually included?
  - Cache: small amount of fast, expensive memory
    - L1 (level 1) cache: usually on the CPU chip
    - L2 & L3 cache: off-chip, made of SRAM
  - Main memory: medium-speed, medium price memory (DRAM)
  - Disk: many gigabytes of slow, cheap, non-volatile storage
- Memory manager handles the memory hierarchy

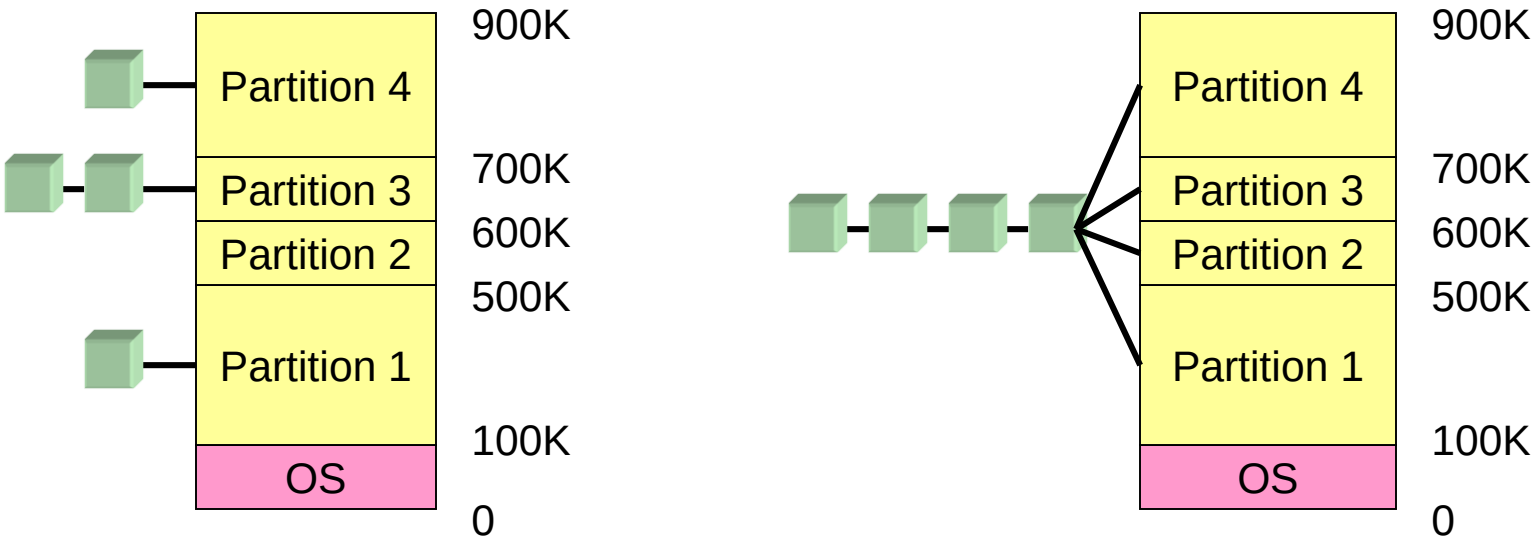
Basic memory management

- Components include
  - Operating system (perhaps with device drivers)
  - Single process
- Goal: lay these out in memory
  - Memory protection may not be an issue (only one program)
  - Flexibility may still be useful (allow OS changes, etc.)
- No swapping or paging



Fixed partitions: multiple programs

- Fixed memory partitions
  - Divide memory into fixed spaces
  - Assign a process to a space when it's free
- Mechanisms
  - Separate input queues for each partition
  - Single input queue: better ability to optimize CPU usage

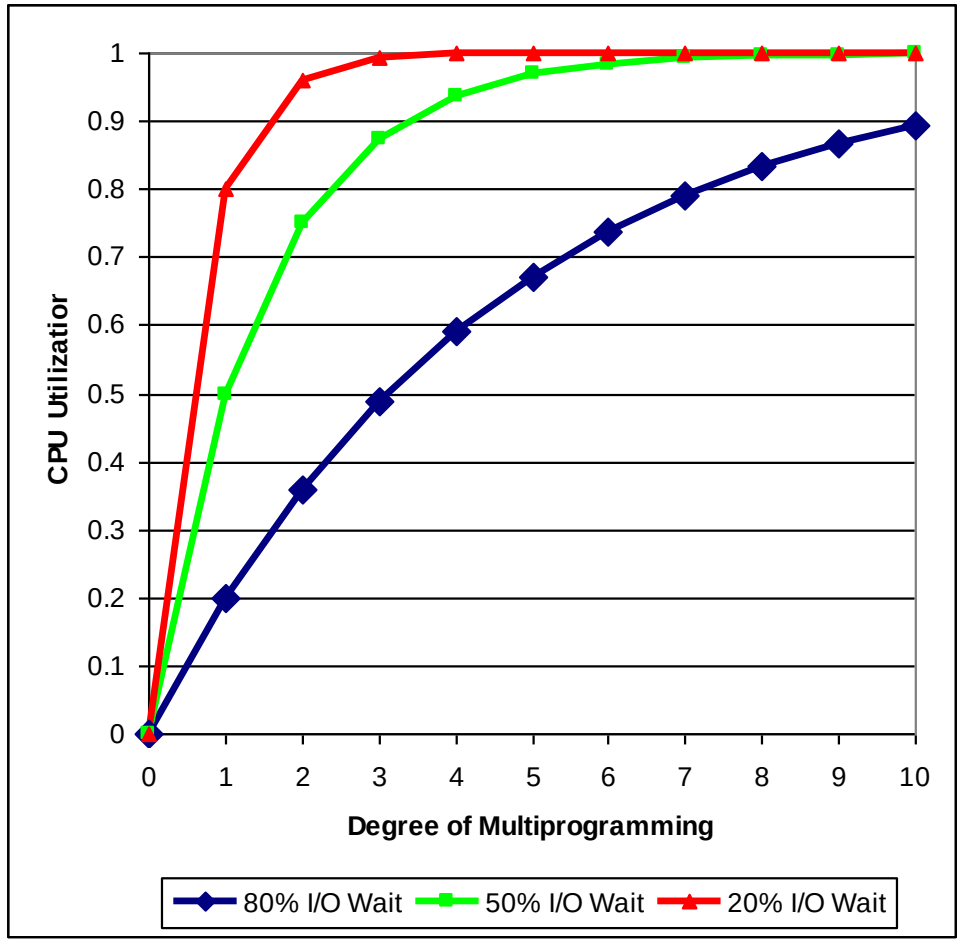


How many programs is enough?

- Several memory partitions (fixed or variable size)
- Lots of processes wanting to use the CPU
- Tradeoff
  - More processes utilize the CPU better
  - Fewer processes use less memory (cheaper!)
- How many processes do we need to keep the CPU fully utilized?
  - This will help determine how much memory we need
  - Is this still relevant with memory costing $150/GB?

Modeling multiprogramming

- More I/O wait means less processor utilization
  - At 20% I/O wait, 3–4 processes fully utilize CPU
  - At 80% I/O wait, even 10 processes aren't enough
- This means that the OS should have more processes if they're I/O bound
- More processes => memory management & protection more important!

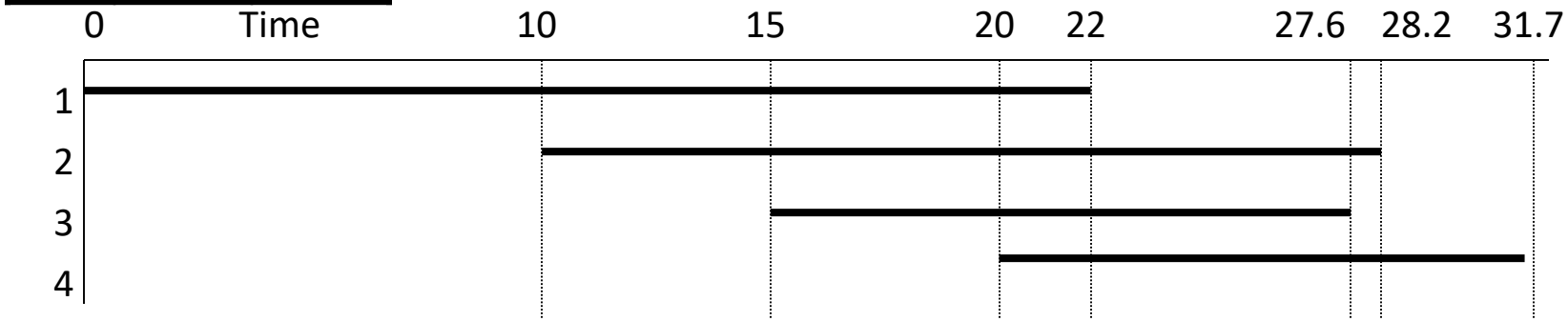


Multiprogrammed system performance

- Arrival and work requirements of 4 jobs
- CPU utilization for 1–4 jobs with 80% I/O wait
- Sequence of events as jobs arrive and finish
  - Numbers show amount of CPU time jobs get in each interval
  - More processes => better utilization, less time per process

Job	Arrival time	CPU needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

	1	2	3	4
CPU idle	0.80	0.64	0.51	0.41
CPU busy	0.20	0.36	0.49	0.59
CPU/process	0.20	0.18	0.16	0.15

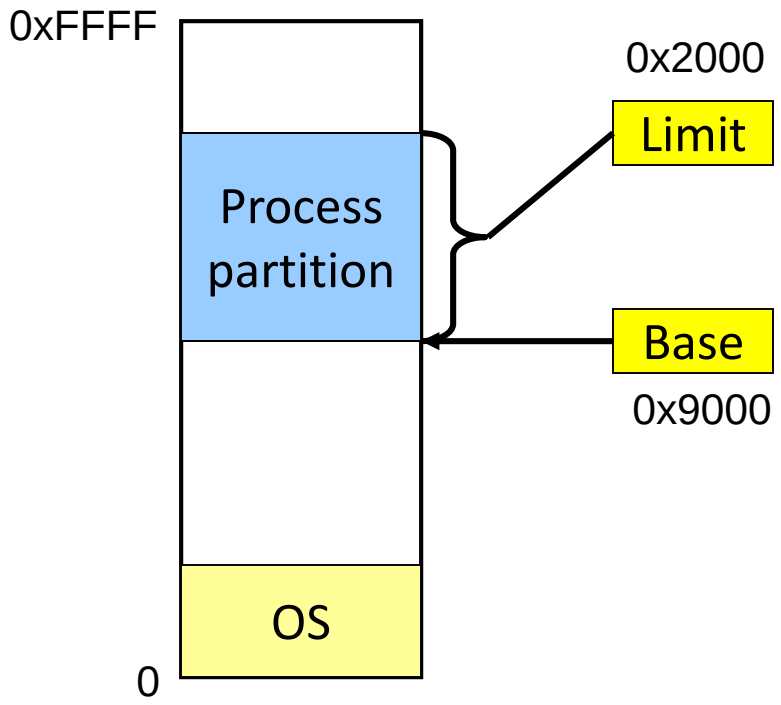


Memory and multiprogramming

- Memory needs two things for multiprogramming
  - Relocation
  - Protection
- The OS cannot be certain where a program will be loaded in memory
  - Variables and procedures can’t use absolute locations in memory
  - Several ways to guarantee this
- The OS must keep processes’ memory separate
  - Protect a process from other processes reading or modifying its own memory
  - Protect a process from modifying its own memory in undesirable ways (such as writing to program code)

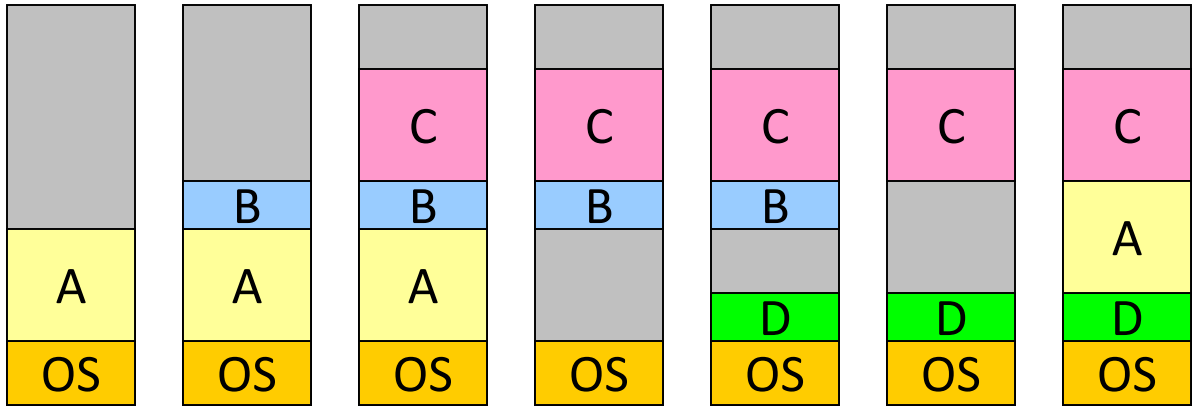
Base and limit registers

- Special CPU registers: base & limit
  - Access to the registers limited to system mode
  - Registers contain
    - Base: start of the process's memory partition
    - Limit: length of the process's memory partition
- Address generation
  - Physical address: location in actual memory
  - Logical address: location from the process's point of view
  - Physical address = base + logical address
  - Logical address larger than limit => error



Logical address: 0x1204
Physical address:
0x1204+0x9000 = 0xa204

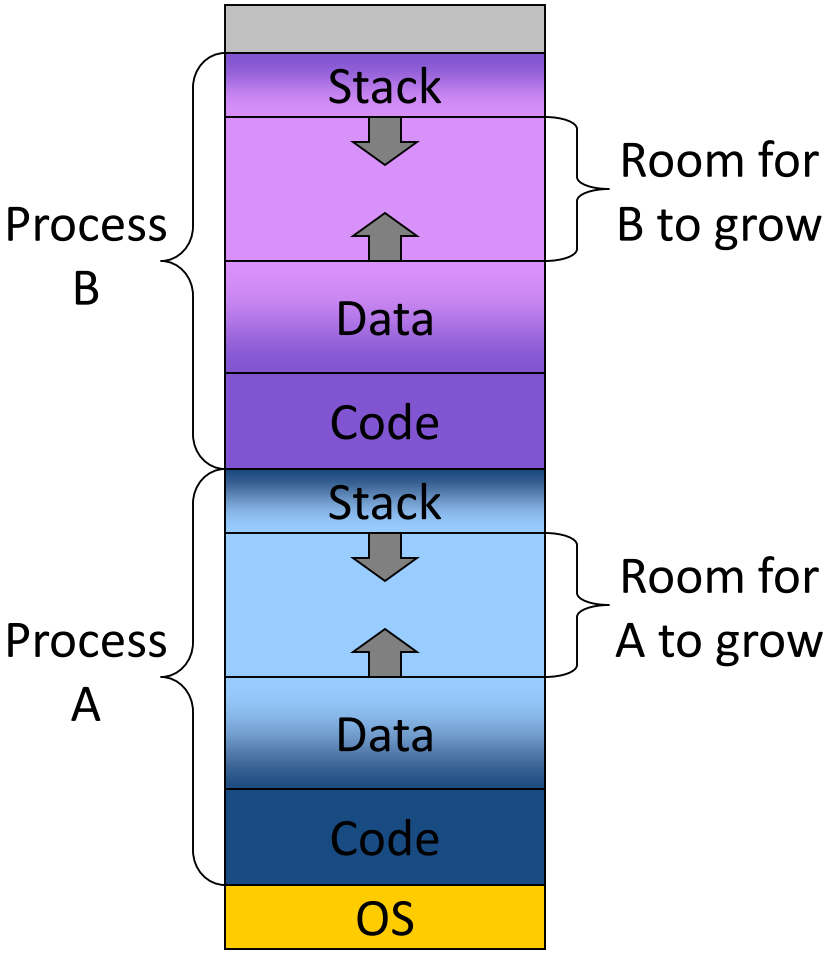
Swapping



- Memory allocation changes as
  - Processes come into memory
  - Processes leave memory
    - Swapped to disk
    - Complete execution
- Gray regions are unused memory

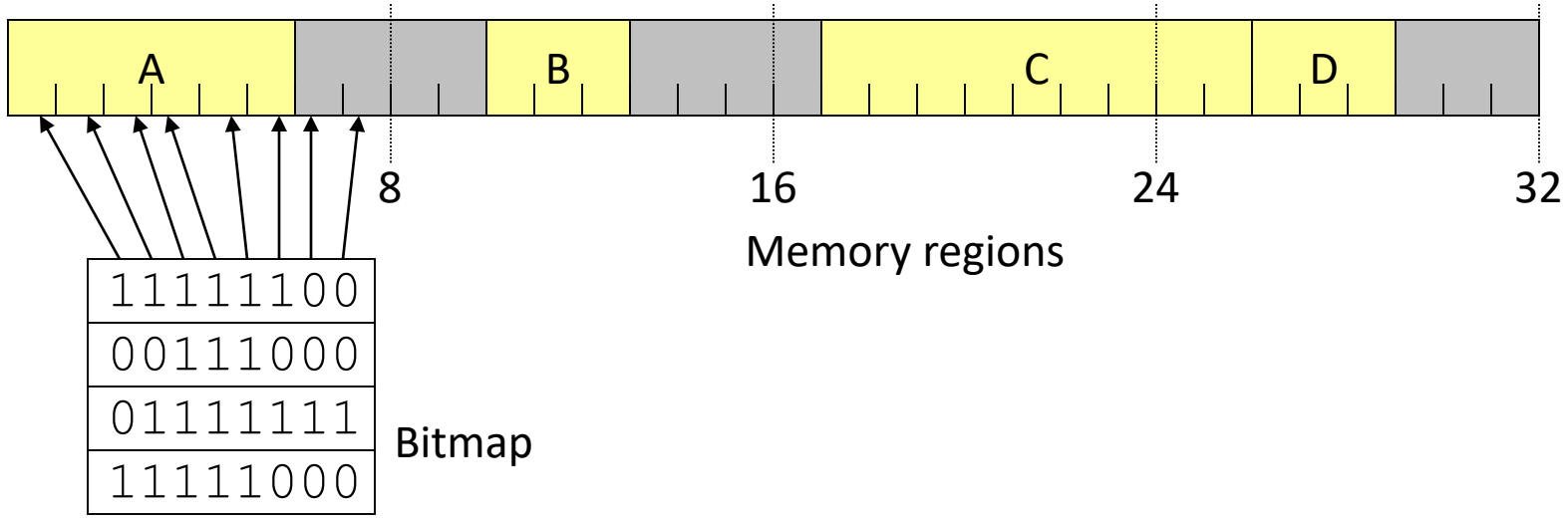
Swapping: leaving room to grow

- Need to allow for programs to grow
  - Allocate more memory for data
  - Larger stack
- Handled by allocating more space than is necessary at the start
  - Inefficient: wastes memory that's not currently in use
  - What if the process requests too much memory?



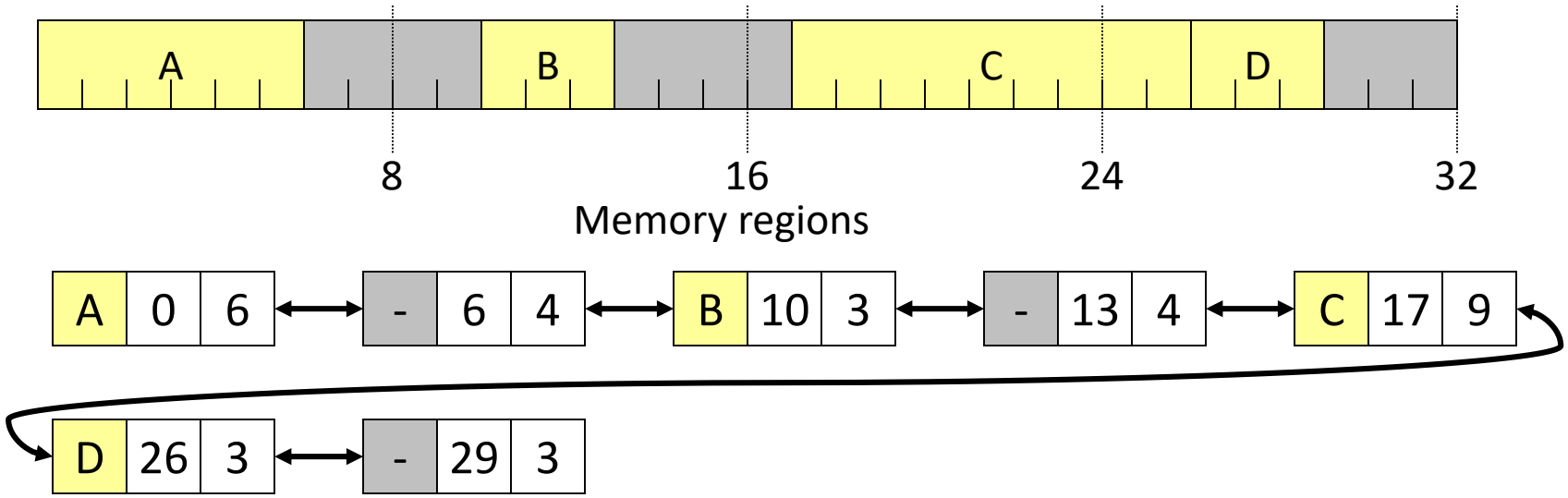
Tracking memory usage: bitmaps

- Keep track of free / allocated memory regions with a bitmap
  - One bit in map corresponds to a fixed-size region of memory
  - Bitmap is a constant size for a given amount of memory regardless of how much is allocated at a particular time
- Chunk size determines efficiency
  - At 1 bit per 4KB chunk, we need just 256 bits (32 bytes) per MB of memory
  - For smaller chunks, we need more memory for the bitmap
  - Can be difficult to find large contiguous free areas in bitmap



Tracking memory usage: linked lists

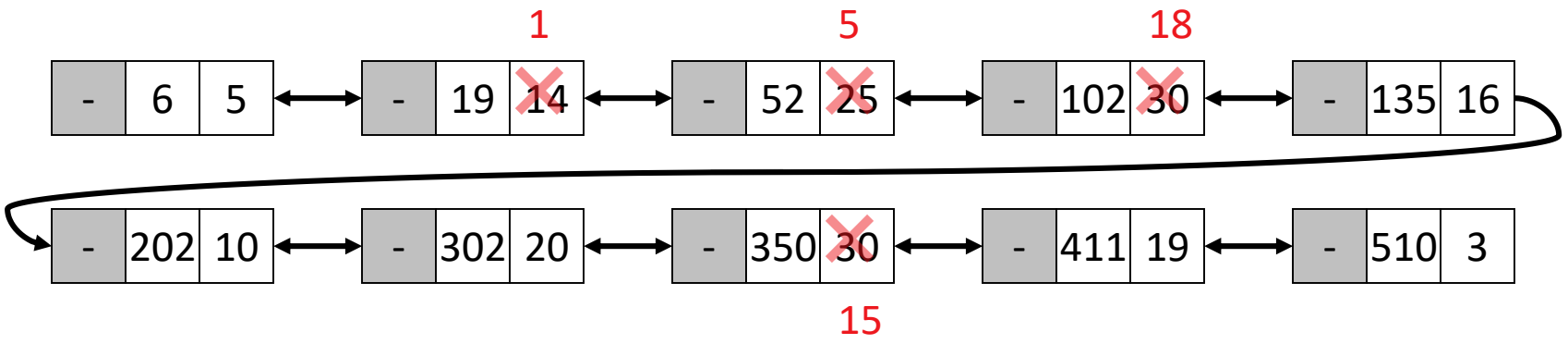
- Keep track of free / allocated memory regions with a linked list
  - Each entry in the list corresponds to a contiguous region of memory
  - Entry can indicate either allocated or free (and, optionally, owning process)
  - May have separate lists for free and allocated areas
- Efficient if chunks are large
  - Fixed-size representation for each region
  - More regions => more space needed for free lists



Allocating memory

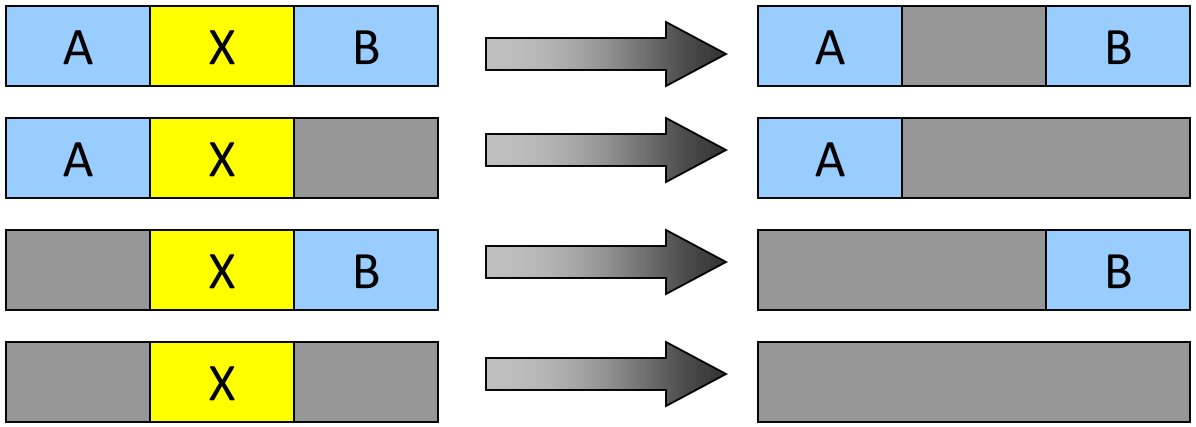
- Search through region list to find a large enough space
- Suppose there are several choices: which one to use?
  - First fit: the first suitable hole on the list
  - Next fit: the first suitable after the previously allocated hole
  - Best fit: the smallest hole that is larger than the desired region (wastes least space?)
  - Worst fit: the largest available hole (leaves largest fragment)
- Option: maintain separate queues for different-size holes

Allocate 20 blocks first fit
Allocate 12 blocks next fit
Allocate 13 blocks best fit
Allocate 15 blocks worst fit



Freeing memory

- Allocation structures must be updated when memory is freed
- Easy with bitmaps: just set the appropriate bits in the bitmap
- Linked lists: modify adjacent elements as needed
  - Merge adjacent free regions into a single region
  - May involve merging two regions with the just-freed area



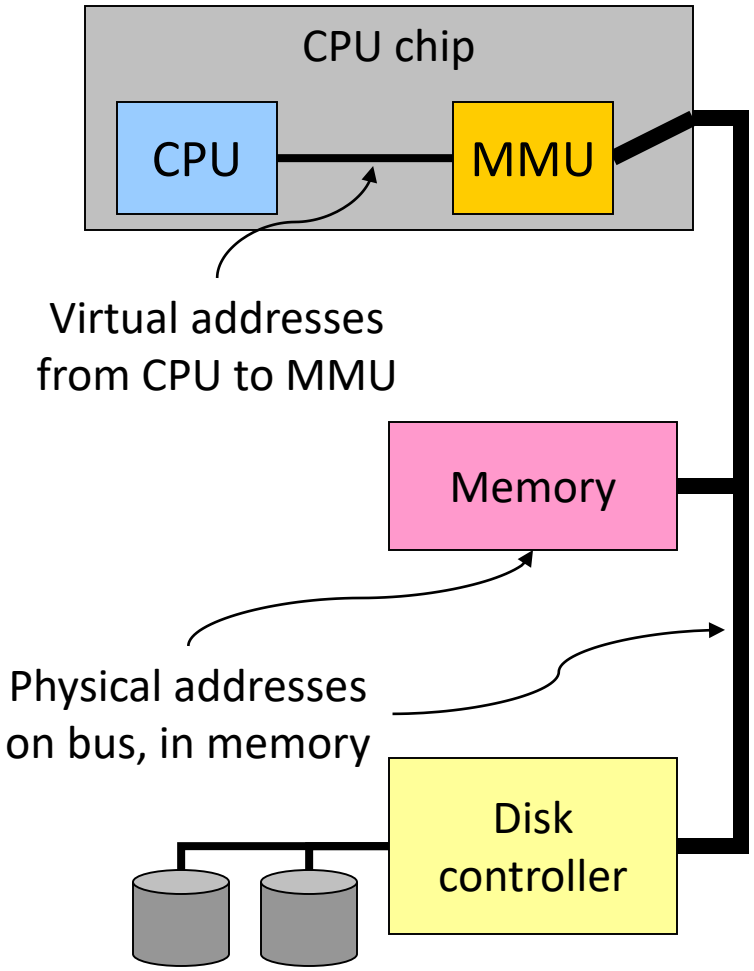
Limitations of swapping

- Problems with swapping
  - Process must fit into physical memory (impossible to run larger processes)
  - Memory becomes fragmented
    - External fragmentation: lots of small free areas
    - Compaction needed to reassemble larger free areas
  - Processes are either in memory or on disk: half and half doesn’t do any good
- Overlays solved the first problem
  - Bring in pieces of the process over time (typically data)
  - Still doesn’t solve the problem of fragmentation or partially resident processes

Virtual memory

- Basic idea: allow the OS to hand out more memory than exists on the system
- Keep recently used stuff in physical memory
- Move less recently used stuff to disk
- Keep all of this hidden from processes
  - Processes still see an address space from 0 – max address
  - Movement of information to and from disk handled by the OS without process help
- Virtual memory (VM) especially helpful in multiprogrammed system
  - CPU schedules process B while process A waits for its memory to be retrieved from disk

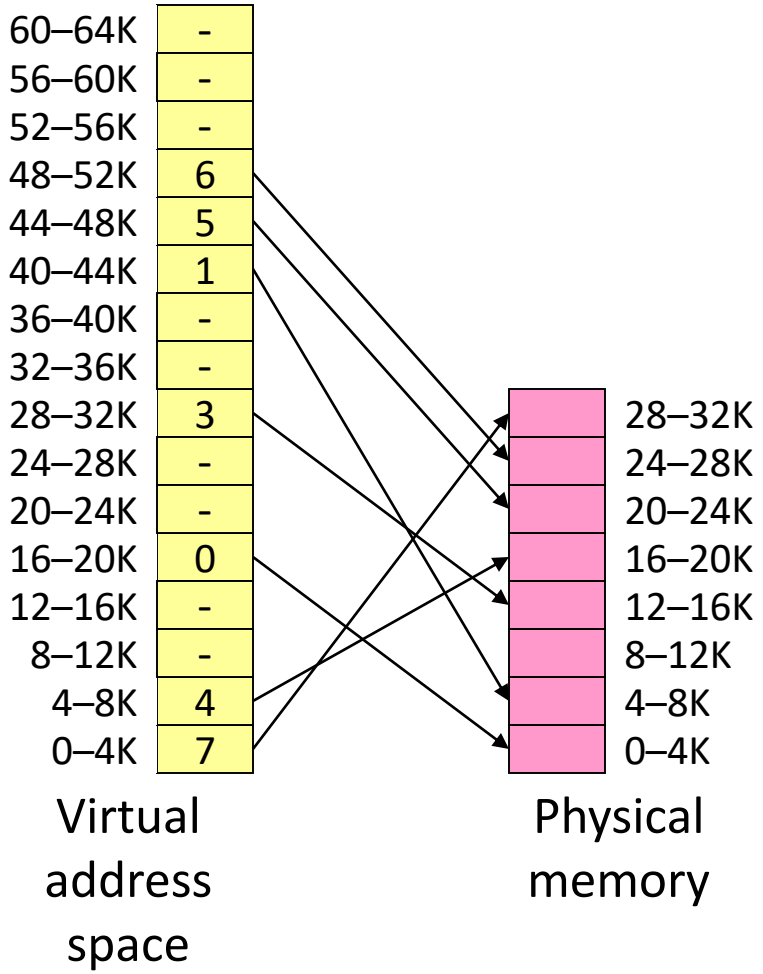
Virtual and physical addresses



- Program uses *virtual addresses*
  - Addresses local to the process
  - Hardware translates virtual address to *physical address*
- Translation done by the ***M**emory **M**anagement **U**nit*
  - Usually on the same chip as the CPU
  - Only physical addresses leave the CPU/MMU chip
- Physical memory indexed by physical addresses

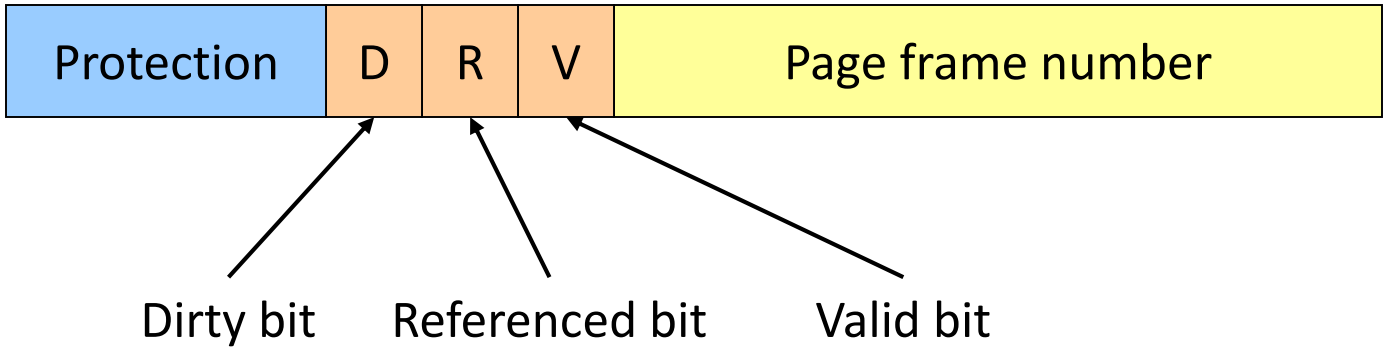
Paging and page tables

- Virtual addresses mapped to physical addresses
  - Unit of mapping is called a *page*
  - All addresses in the same virtual page are in the same physical page
  - *Page table entry* (PTE) contains translation for a single page
- Table translates virtual page number to physical page number
  - Not all virtual memory has a physical page
  - Not every physical page need be used
- Example:
  - 64 KB virtual memory
  - 32 KB physical memory



What's in a page table entry?

- Each entry in the page table contains
  - Valid bit: set if this logical page number has a corresponding physical frame in memory
    - If not valid, remainder of PTE is irrelevant
  - Page frame number: page in physical memory
  - Referenced bit: set if data on the page has been accessed
  - Dirty (modified) bit :set if data on the page has been modified
  - Protection information

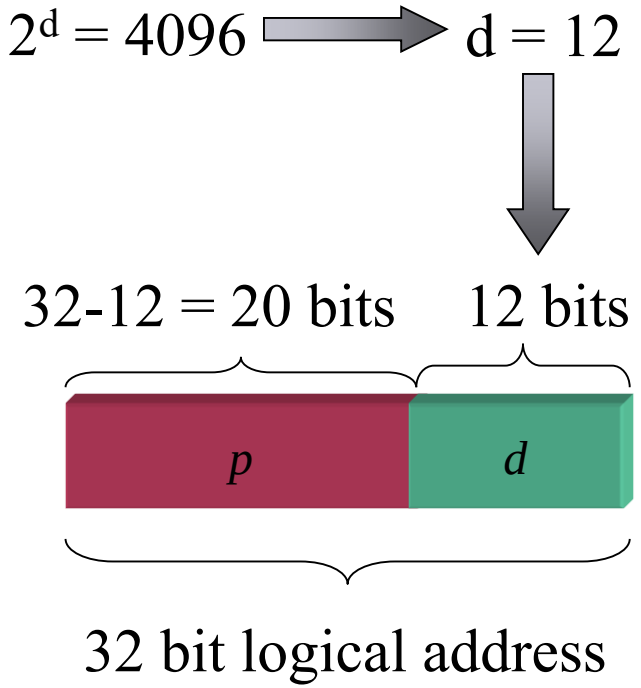


Mapping logical => physical address

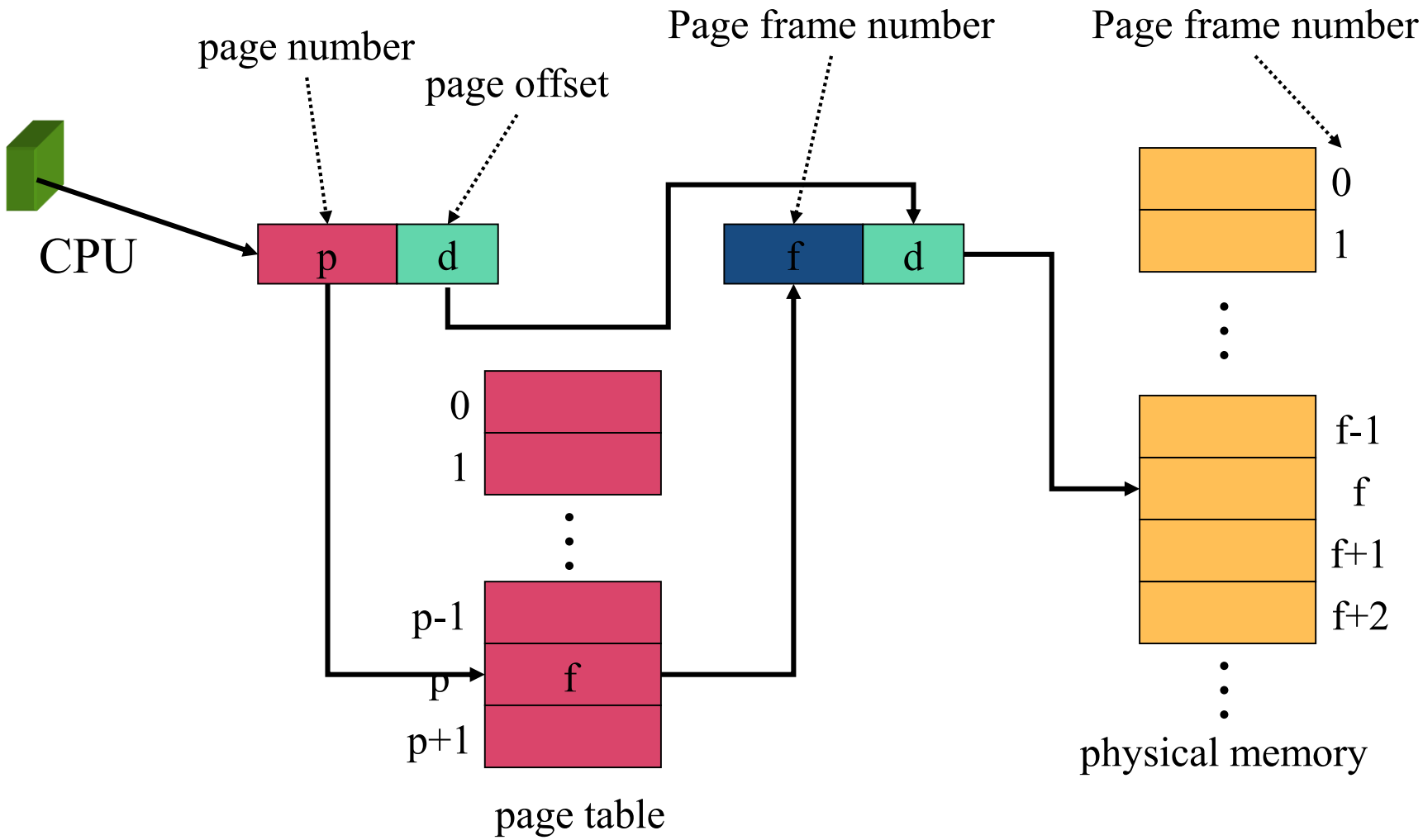
- Split address from CPU into two pieces
  - Page number (*p*)
  - Page offset (*d*)
- Page number
  - Index into page table
  - Page table contains base address of page in physical memory
- Page offset
  - Added to base address to get actual physical memory address
- Page size = 2^d bytes

Example:

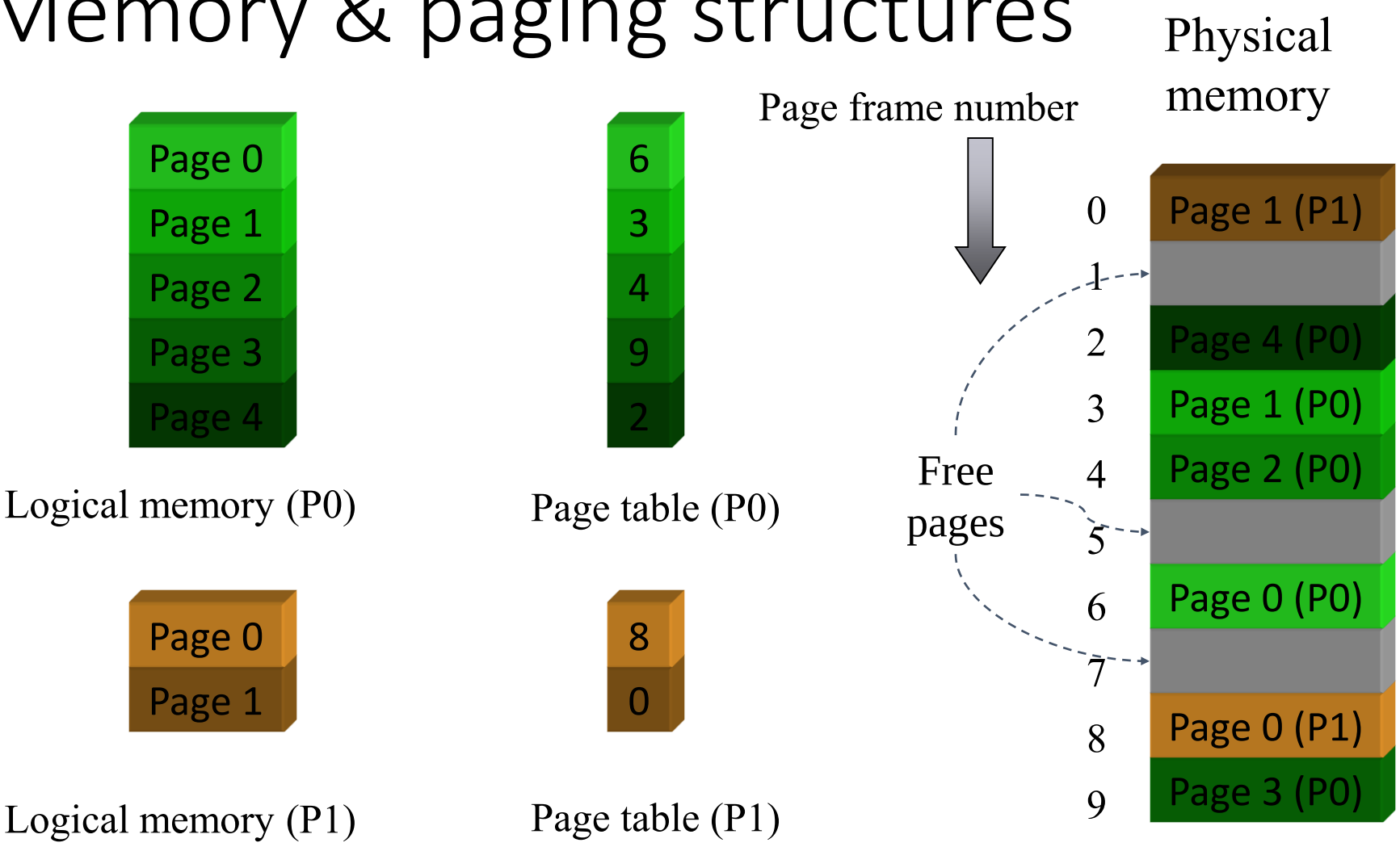
- 4 KB (=4096 byte) pages
- 32 bit logical addresses



Address translation architecture

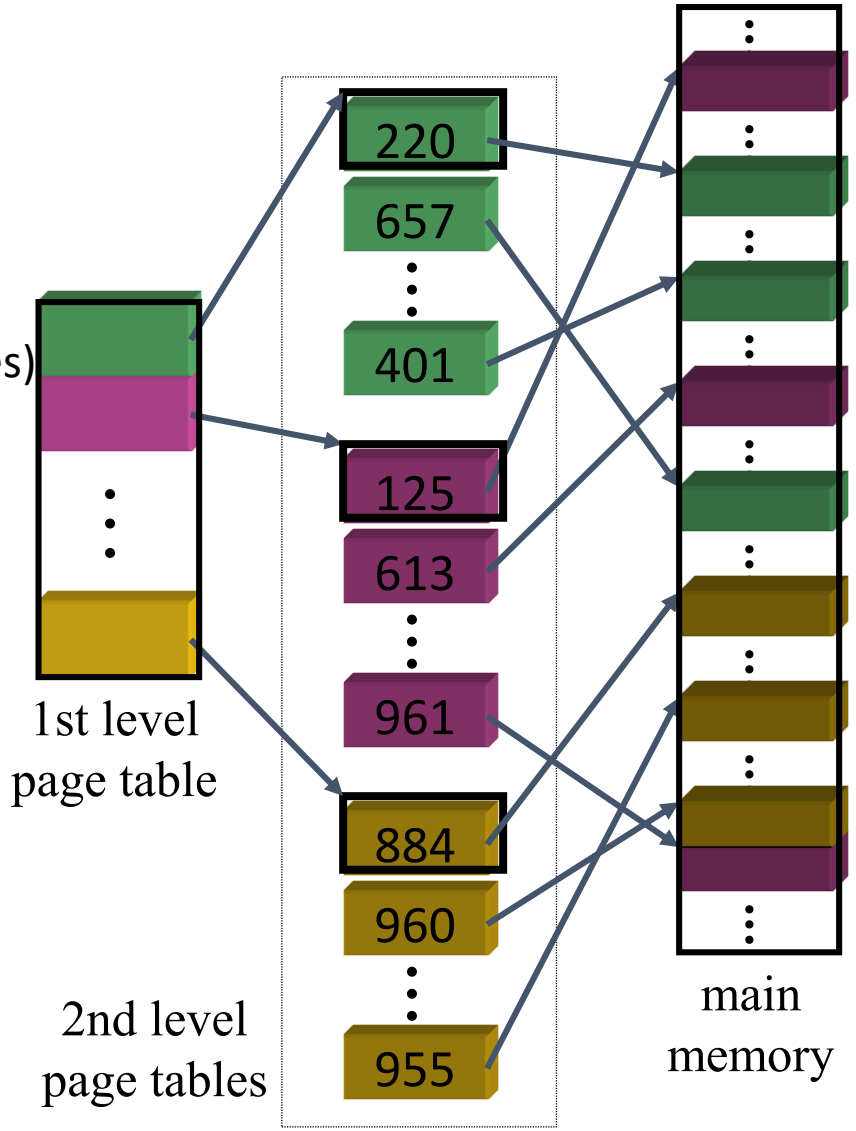


Memory & paging structures



Two-level page tables

- Problem: page tables can be too large
  - 2^{32} bytes in 4KB pages need 1 million PTEs
- Solution: use multi-level page tables
  - "Page size" in first page table is large (megabytes)
  - PTE marked invalid in first page table needs no 2nd level page table
- 1st level page table has pointers to 2nd level page tables
- 2nd level page table has actual physical page numbers in it

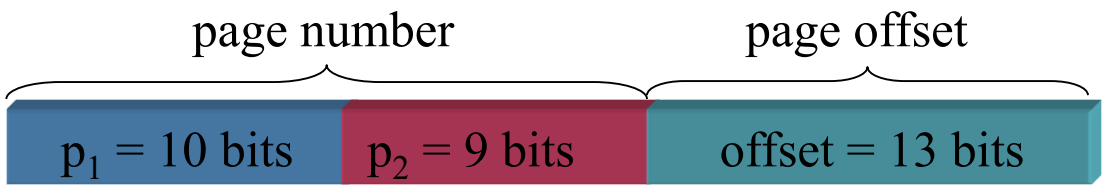


More on two-level page tables

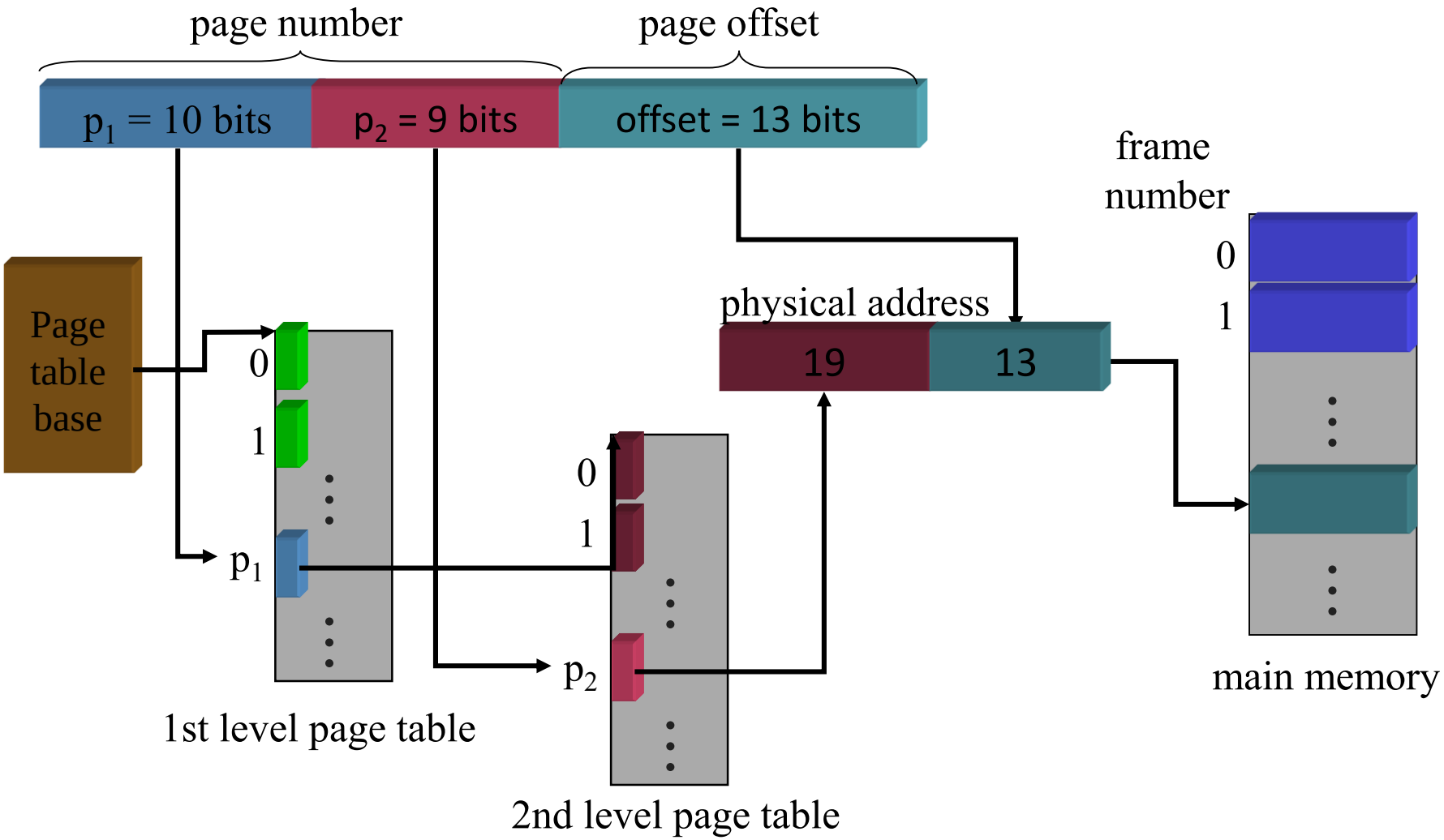
- Tradeoffs between 1st and 2nd level page table sizes
  - Total number of bits indexing 1st and 2nd level table is constant for a given page size and logical address length
  - Tradeoff between number of bits indexing 1st and number indexing 2nd level tables
    - More bits in 1st level: fine granularity at 2nd level
    - Fewer bits in 1st level: maybe less wasted space?
- All addresses in table are physical addresses
- Protection bits kept in 2nd level table

Two-level paging: example

- System characteristics
  - 8 KB pages
  - 32-bit logical address divided into 13 bit page offset, 19 bit page number
- Page number divided into:
  - 10 bit page number
  - 9 bit page offset
- Logical address looks like this:
  - p_1 is an index into the 1st level page table
  - p_2 is an index into the 2nd level page table pointed to by p_1



2-level address translation example

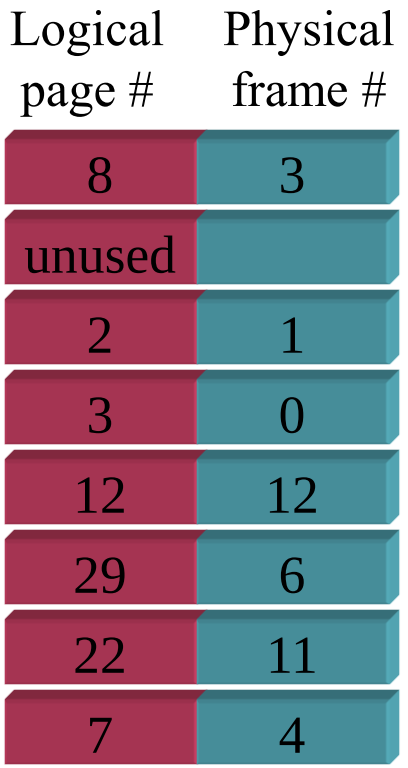


Implementing page tables in hardware

- Page table resides in main (physical) memory
- CPU uses special registers for paging
  - Page table base register (PTBR) points to the page table
  - Page table length register (PTLR) contains length of page table: restricts maximum legal logical address
- Translating an address requires two memory accesses
  - First access reads page table entry (PTE)
  - Second access reads the data / instruction from memory
- Reduce number of memory accesses
  - Can't avoid second access (we need the value from memory)
  - Eliminate first access by keeping a hardware cache (called a *translation lookaside buffer* or TLB) of recently used page table entries

Translation Lookaside Buffer (TLB)

- Search the TLB for the desired logical page number
  - Search entries in parallel
  - Use standard cache techniques
- If desired logical page number is found, get frame number from TLB
- If desired logical page number isn't found
  - Get frame number from page table in memory
  - Replace an entry in the TLB with the logical & physical page numbers from this reference



Example TLB

Handling TLB misses

- If PTE isn't found in TLB, OS needs to do the lookup in the page table
- Lookup can be done in hardware or software
- Hardware TLB replacement
  - CPU hardware does page table lookup
  - Can be faster than software
  - Less flexible than software, and more complex hardware
- Software TLB replacement
  - OS gets TLB exception
  - Exception handler does page table lookup & places the result into the TLB
  - Program continues after return from exception
  - Larger TLB (lower miss rate) can make this feasible

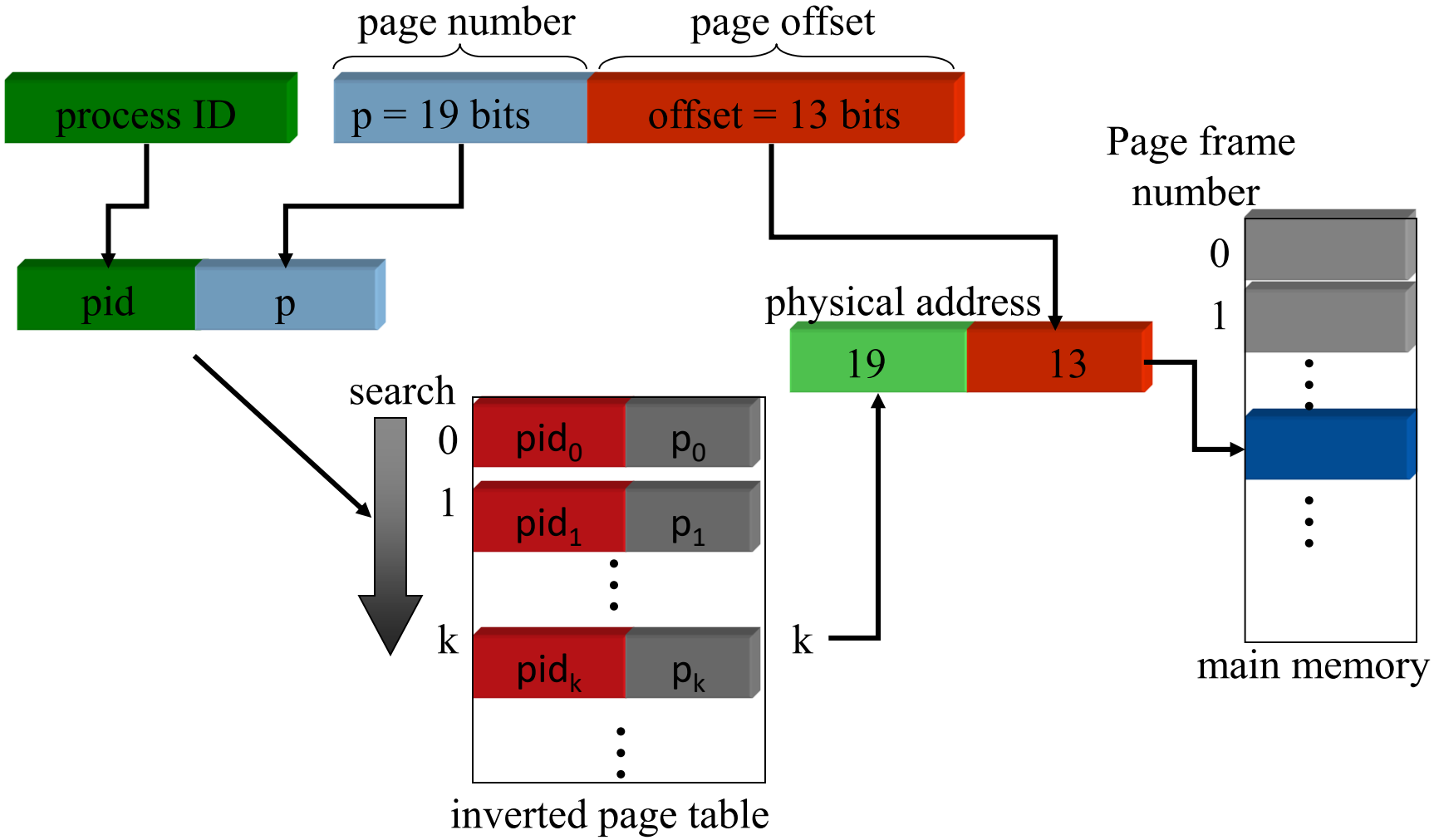
How long do memory accesses take?

- Assume the following times:
  - TLB lookup time = a (often zero - overlapped in CPU)
  - Memory access time = m
- Hit ratio (h) is percentage of time that a logical page number is found in the TLB
  - Larger TLB usually means higher h
  - TLB structure can affect h as well
- Effective access time (an average) is calculated as:
  - EAT = (m + a)h + (m + m + a)(1-h)
  - EAT =a + (2-h)m
- Interpretation
  - Reference always requires TLB lookup, 1 memory access
  - TLB misses also require an additional memory reference

Inverted page table

- Reduce page table size further: keep one entry for each frame in memory
- PTE contains
  - Virtual address pointing to this frame
  - Information about the process that owns this page
- Search page table by
  - Hashing the virtual page number and process ID
  - Starting at the entry corresponding to the hash result
  - Search until either the entry is found or a limit is reached
- Page frame number is index of PTE
- Improve performance by using more advanced hashing algorithms

Inverted page table architecture



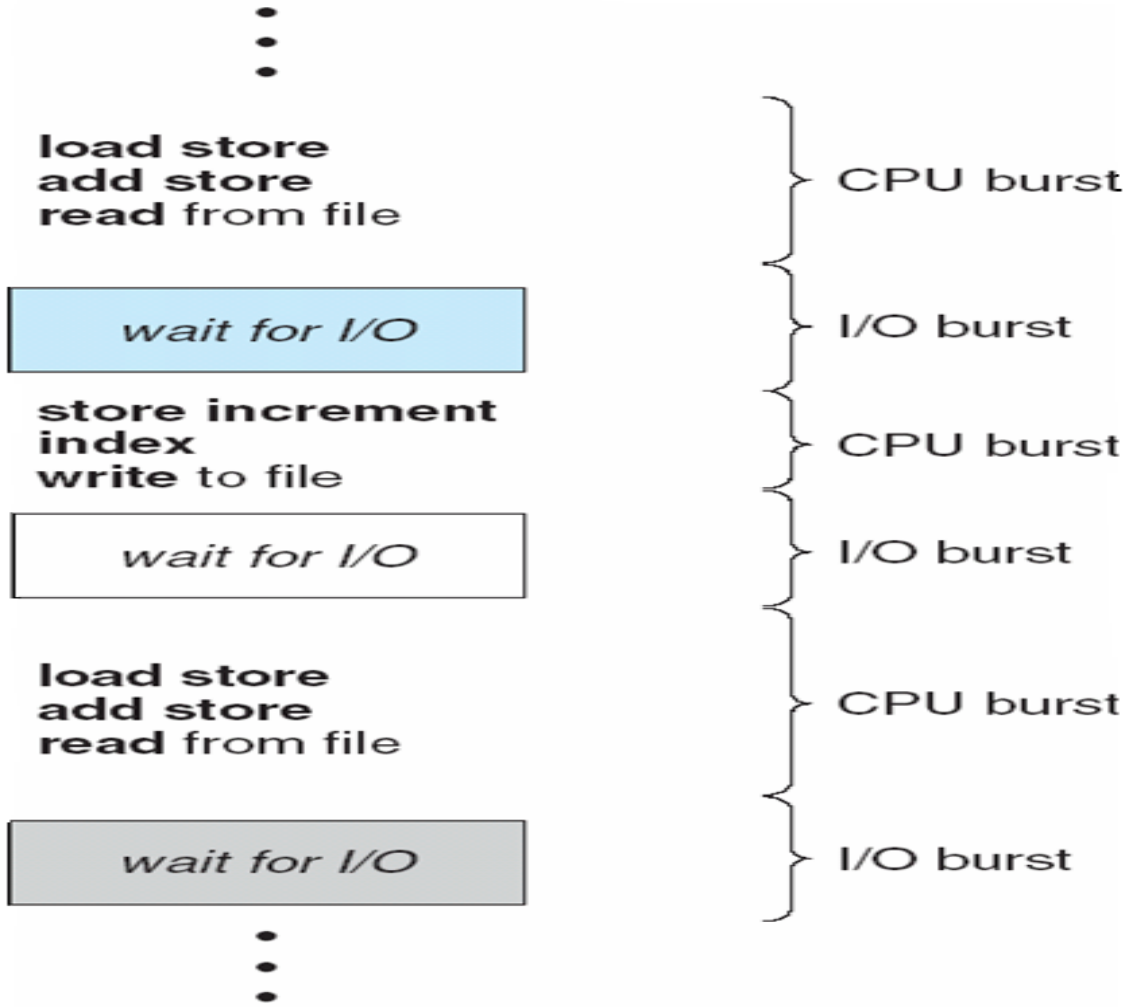
ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

CPU Scheduling

The CPU-I/O Burst Cycle

- Maximum CPU utilization obtained with multiprogramming.
- CPU-I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait.
- Each cycle consists of a CPU burst followed by a (usually longer) I/O burst.
- A process usually terminates on a CPU burst.
- CPU burst distribution is of main concern.

Alternating sequence of CPU and I/O Bursts

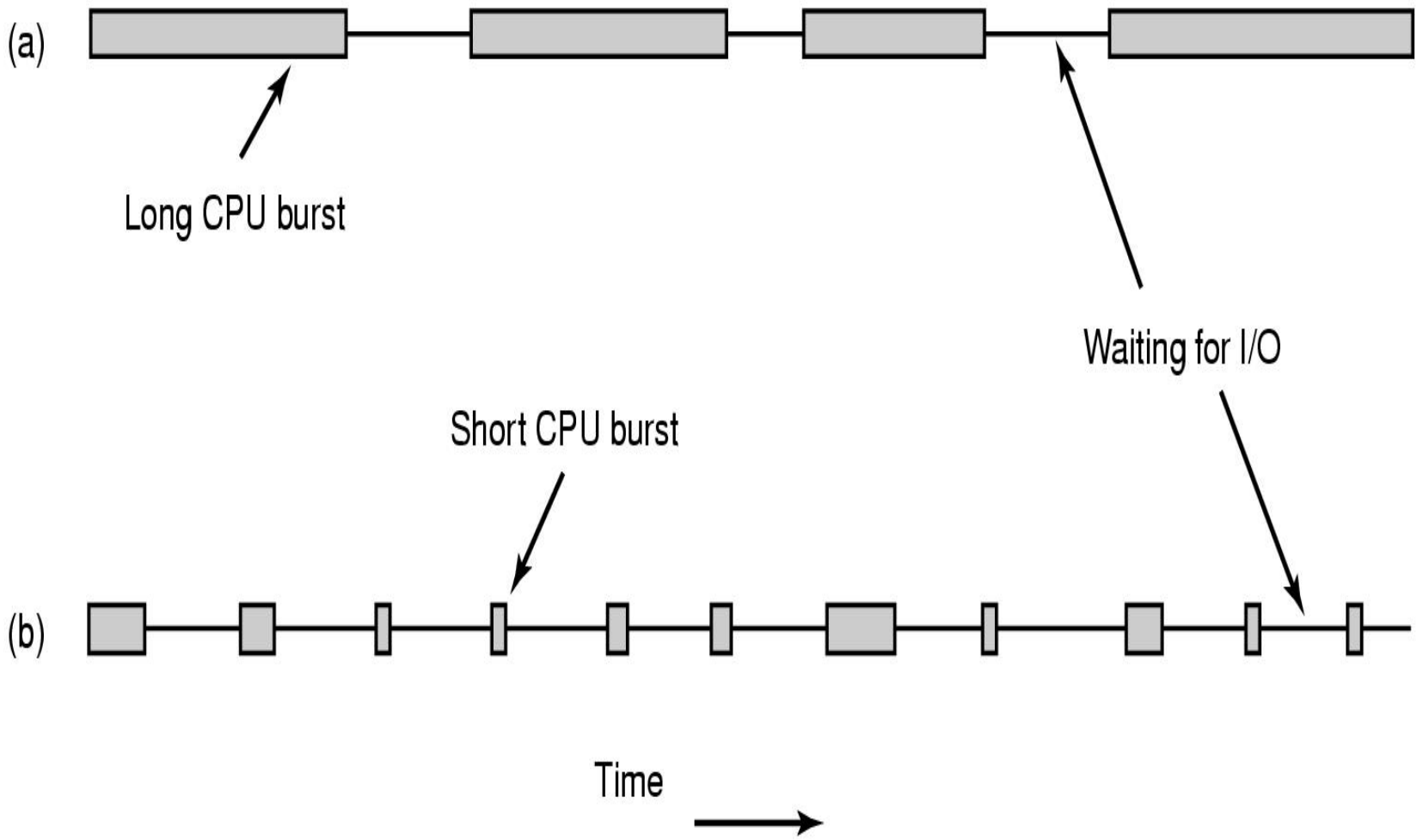


ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

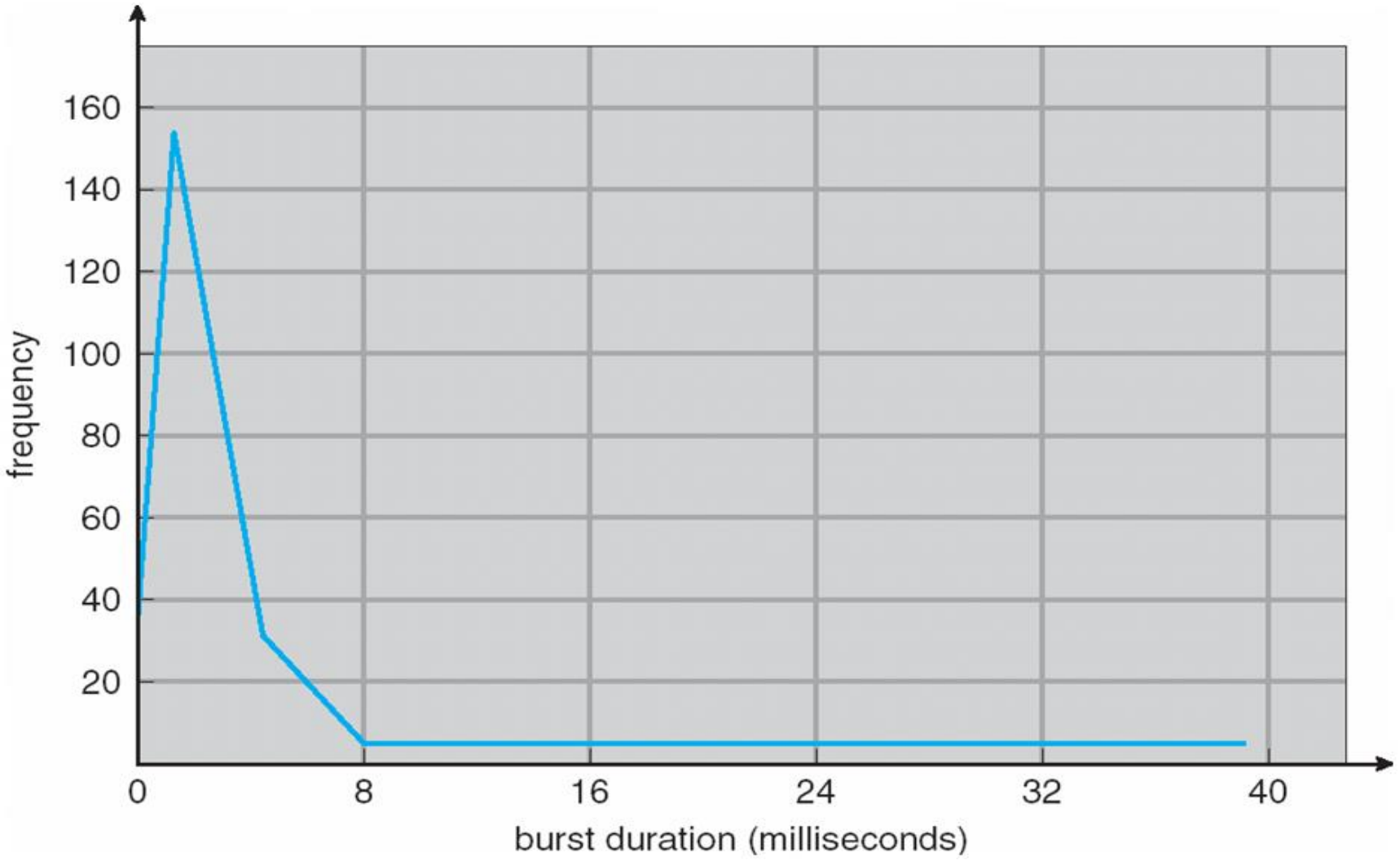
Basic Concepts

- Burst/Service time = total processor time needed in one CPU-I/O burst cycle.
- Jobs with long CPU burst time are CPU-bound jobs and are also referred to as “long jobs”.
- Jobs with short CPU burst time are IO-bound jobs and are also referred to as “short jobs”.
- CPU-bound processes have longer CPU bursts than I/O-bound processes.

Long vs. Short CPU Burst time



Histogram of CPU burst Times



Scheduling Goals of Different Systems

All systems

- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems

- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems

- Response time - respond to requests quickly
- Proportionality - meet users' expectations

Real-time systems

- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Uniprocessor Scheduling

- We concentrate on the problem of scheduling the usage of a single processor among all the existing ready processes in the system (short-term scheduling).
- The goal is to achieve:
  - High processor utilization.
  - High throughput
    - number of processes completed per unit time.
  - Low turnaround/response time.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Scheduling Criteria

- CPU utilization – keeping CPU as busy as possible.
- Throughput – # of processes that complete their execution per time unit.
- Turnaround time – amount of time to execute a particular process from start to end.
- Waiting time – amount of time a process has been waiting in the ready queue.
- Response time – amount of time it takes from when a request was submitted until the first response is produced (for time-sharing environment).

Classification of Scheduling Criteria

- User-oriented:
  - Response Time: Elapsed time from the submission of a request to the beginning of response.
  - Turnaround Time: Elapsed time from the submission of a process to its completion.
- System-oriented:
  - CPU utilization
  - Throughput: number of process completed per unit time.
  - Fairness

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Optimization Criteria

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

CPU (short-term) Scheduler

- Selects from among the processes that are ready to execute, and allocates the CPU to one of them.
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state.
  2. Switches from running to ready state.
  3. Switches from waiting to ready.
  4. Terminates.
- Scheduling under 1 and 4 is *nonpreemptive*.
- All other scheduling is *preemptive.*

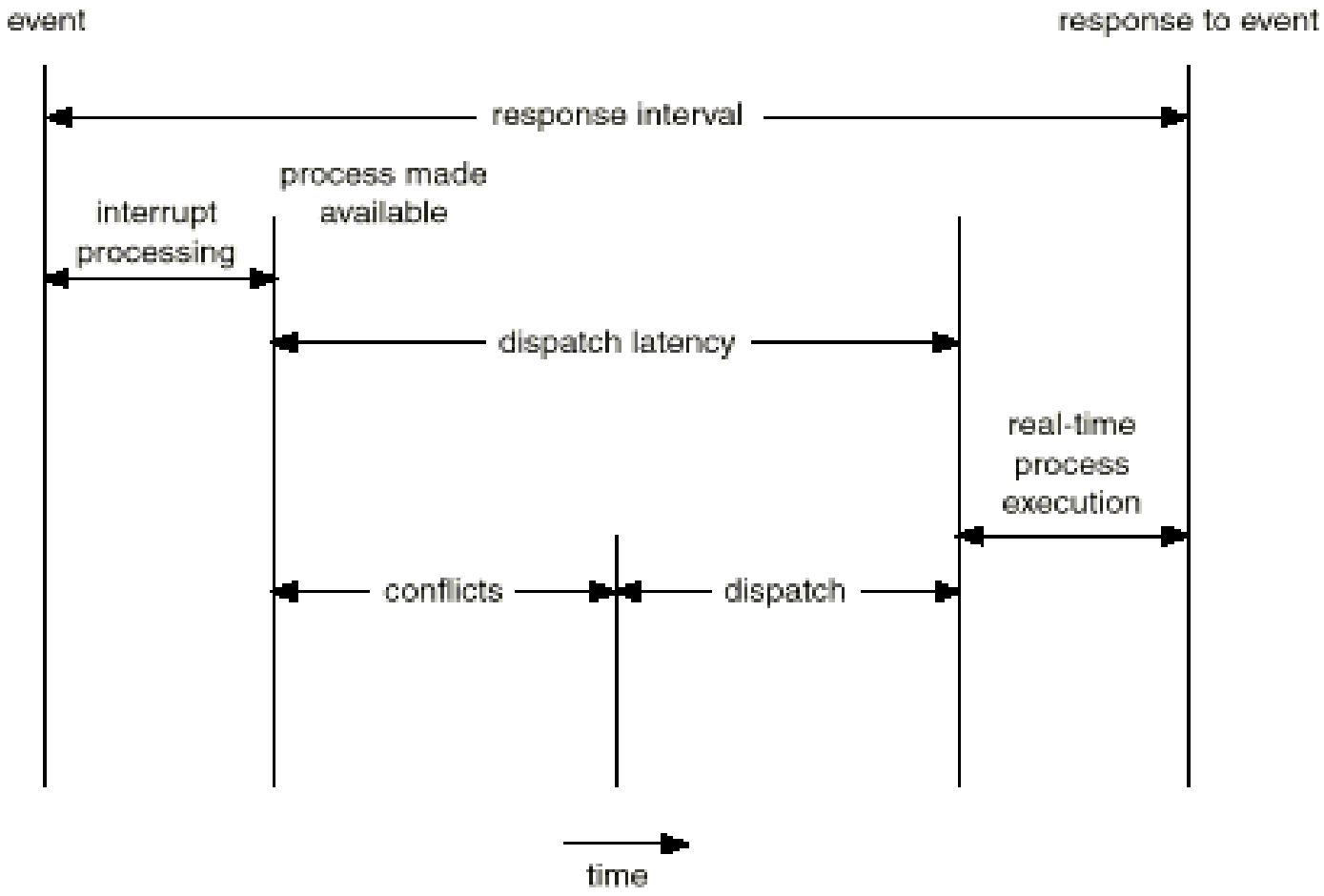
Characterization of Scheduling Policies

- The selection function: determines which process in the ready queue is selected next for execution.
- The decision mode: specifies the instants in time at which the selection function is exercised.
  - Nonpreemptive:
    - Once a process is in the running state, it will continue until it terminates or blocks itself for I/O.
  - Preemptive:
    - Currently running process may be interrupted and moved to the Ready state by the OS.
    - Allows for better service since any one process cannot monopolize the processor for very long.

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.

Dispatch Latency

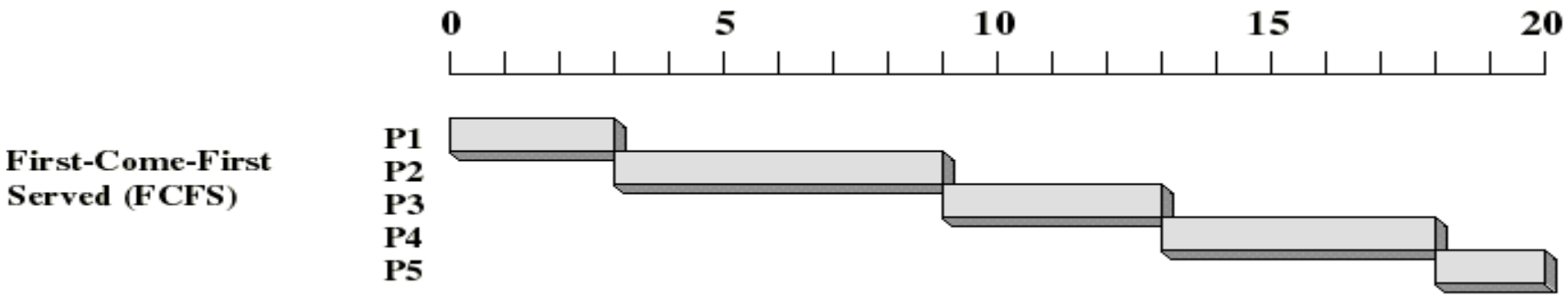


First Come First Served (FCFS)

- Selection function: the process that has been waiting the longest in the ready queue – hence called First-Come First-Served (FCFS).
- Decision mode: Nonpreemptive.

FCFS Example

Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



A Simpler FCFS Example

Process	Burst Time
P_1	24
P_2	3
P_3	3

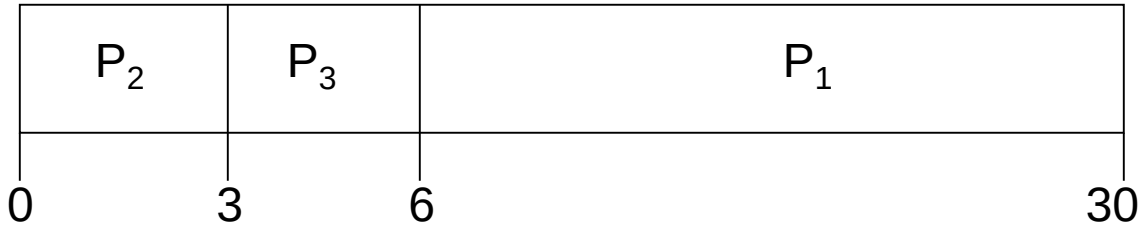
- Suppose that the processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- *Convoy effect:* short process behind long process:
  - Consider one CPU-bound and many I/O-bound processes.

A Twist on the FCFS Example

Suppose that 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.

FCFS Drawbacks

- A process that does not perform any I/O will monopolize the processor (Convoy Effect).
- Favors CPU-bound processes:
  - I/O-bound processes have to wait until CPU-bound process completes.
  - They may have to wait even when their I/O are completed (poor device utilization).
  - We could have kept the I/O devices busy by giving a bit more priority to I/O bound processes.

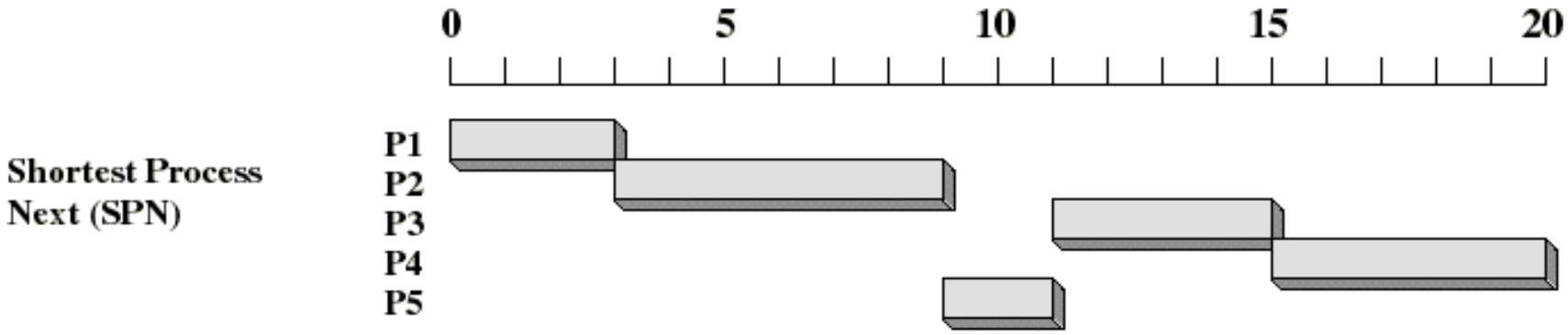
ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Shortest Job First (SJF)

- Selection function: the process with the shortest expected CPU burst time; Need to associate with each process the length of its next CPU burst.
- Decision mode: Nonpreemptive.
- Called also Shortest Time First (STF) and Shortest Process Next (SPN).
- I/O bound processes will be picked first.
- SJF is optimal – gives minimum average waiting time for a given set of processes.

Shortest Job First (SJF) Example

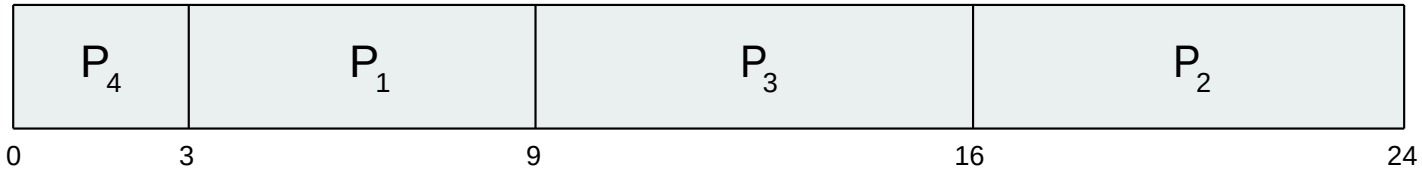
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Simple Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart

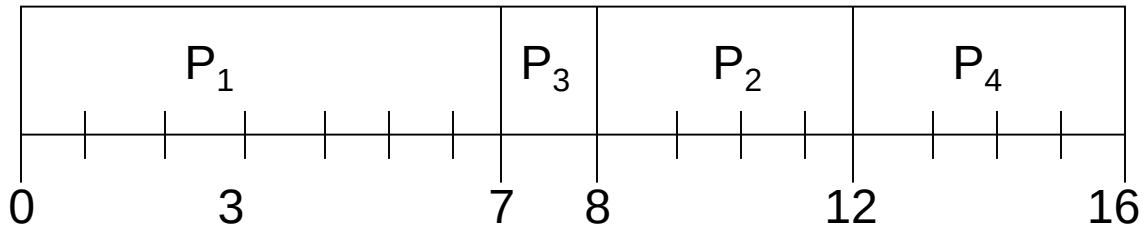


- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Another Example of SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Dynamics of Shortest-Job-First (SJF)

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
- We need to somehow estimate the required processing time (CPU burst time) for each process.

Determining length of next CPU Burst

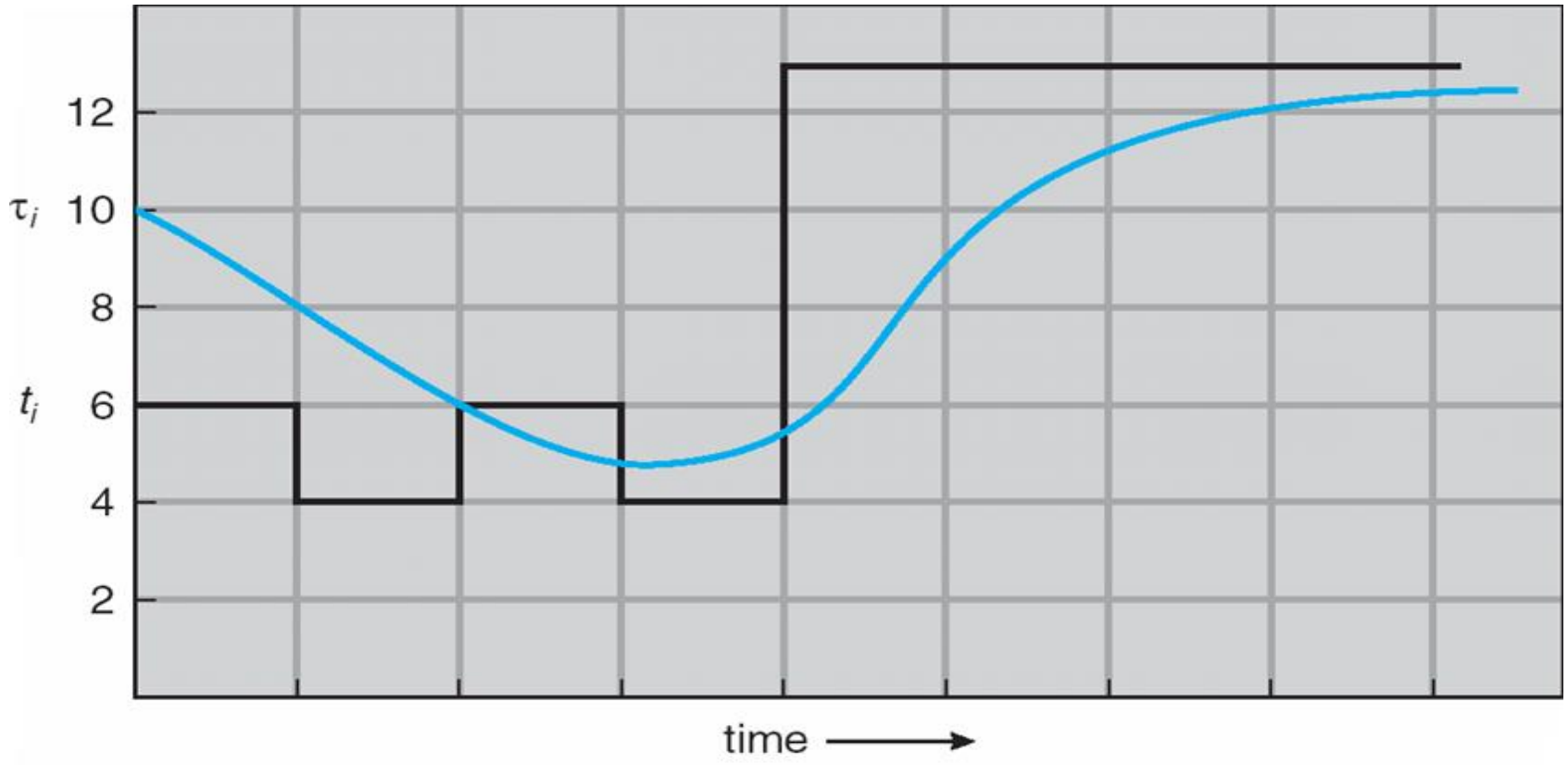
- Can only estimate the length – should be similar to the previous one.
- Can be done by using the length of previous CPU bursts, using exponential averaging:

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define : $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\,\tau_n .$

Examples of Exponential Averaging

- How to set α in $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$?
- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$,
  - Recent history does not count.
- $\alpha = 1$
  - $\tau_{n+1} = t_n$,
  - Only the actual last CPU burst counts.
- Let's be balanced: $\alpha = 0.5$ – See example in next slide.

Prediction of the length of the next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Idea of Exponential Averaging

- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\,\alpha\, t_{n-1} + \ldots$$
$$+(1-\alpha)^j\,\alpha\, t_{n-j} + \ldots$$
$$+(1-\alpha)^{n+1}\,\tau_0$$

- Since both α and $(1-\alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor; term weights are decreasing exponentially.
- Exponential averaging here is better than simple averaging.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Shortest Job First Drawbacks

- Possibility of starvation for longer processes as long as there is a steady supply of shorter processes.
- Lack of preemption is not suited in a time sharing environment:
  - CPU bound process gets lower priority (as it should) but a process doing no I/O could still monopolize the CPU if he is the first one to enter the system.
- SJF implicitly incorporates priorities: shortest jobs are given preferences.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

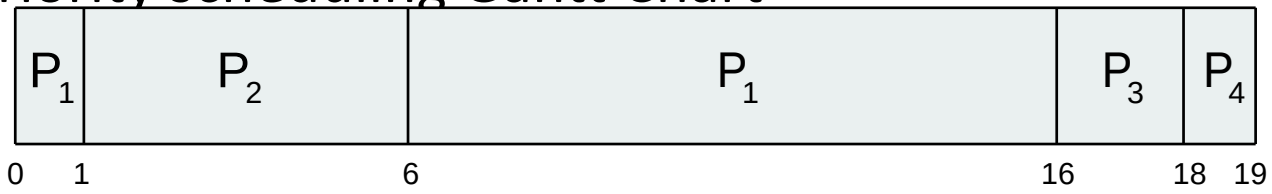
Priority Scheduling

- A priority number is associated with each process.
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority).
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem: Starvation – low priority processes may never execute.
- Solution: **Aging** – as time progresses, increase the priority of the process.

Example of Priority Scheduling

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



- Average waiting time = 8.2

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Advanced CPU Scheduling

Idea of Time Quantum

- Decision mode: preemptive –
  - a process is allowed to run until the set time slice period, called time quantum, is reached.
  - then a clock interrupt occurs and the running process is put on the ready queue.
- How to set the quantum q?

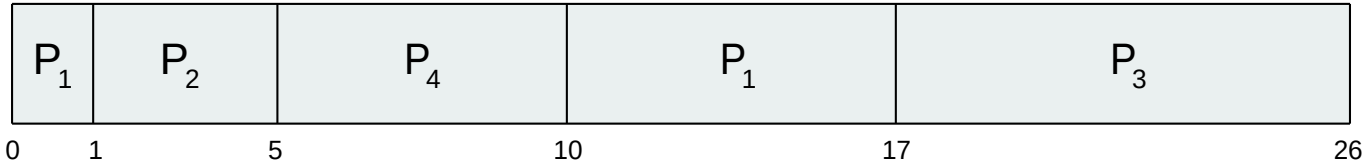
Shortest-Remaining-Job-First (SRJF)

- Associate with each process the length of its next/remaining CPU burst. Use these lengths to schedule the process with the shortest time.
- Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt.
- Called Shortest-Remaining-Job-First (SRJF).

Example of Shortest-remaining-time-first

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

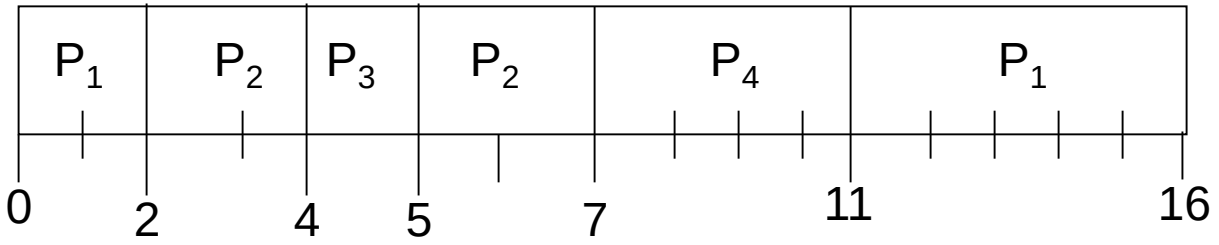


- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5

Another Example of SRJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

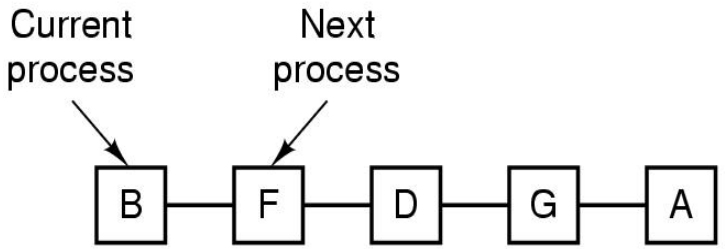
- SRJF (preemptive) with q = 2



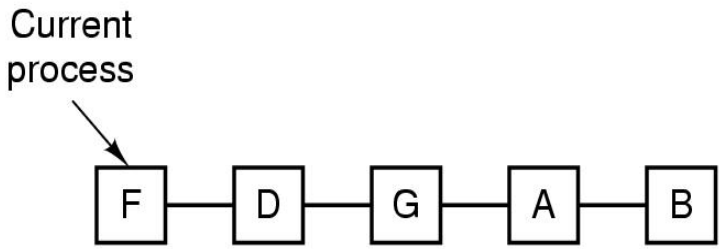
- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Round-Robin (RR)

- Selection function: (initially) same as FCFS.
- Decision mode: preemptive –
  - a process is allowed to run until the time slice period, called time quantum, is reached.
  - then a clock interrupt occurs and the running process is put at the end of the ready queue.



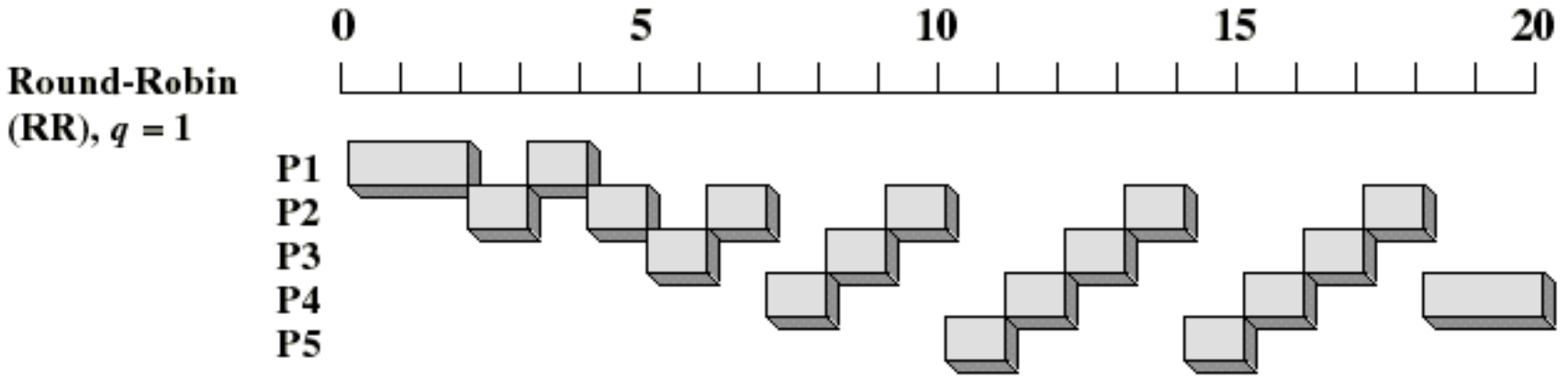
(a)



(b)

Round-Robin (RR) Example

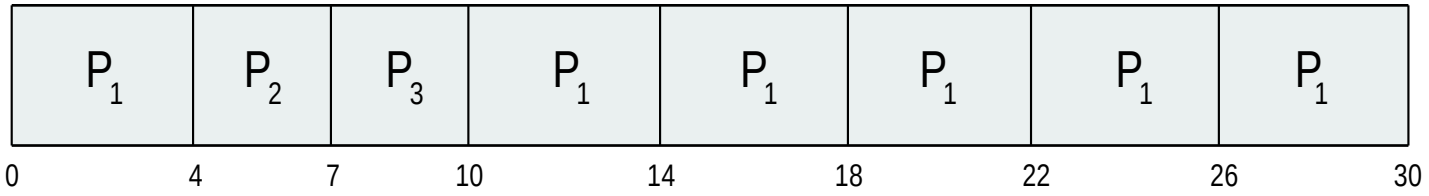
Process	Arrival Time	Service Time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2



Example of RR with Time Quantum = 4

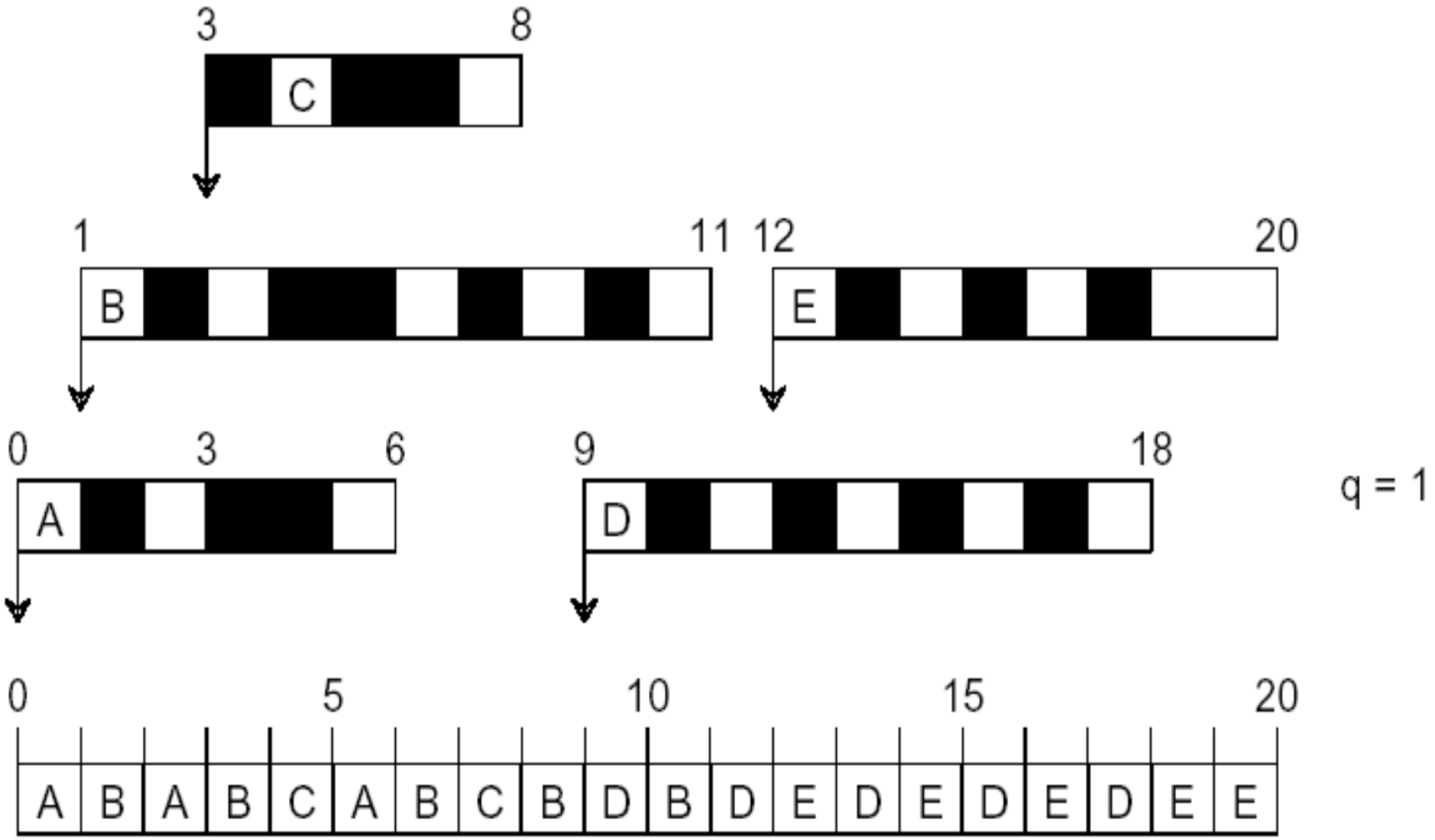
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

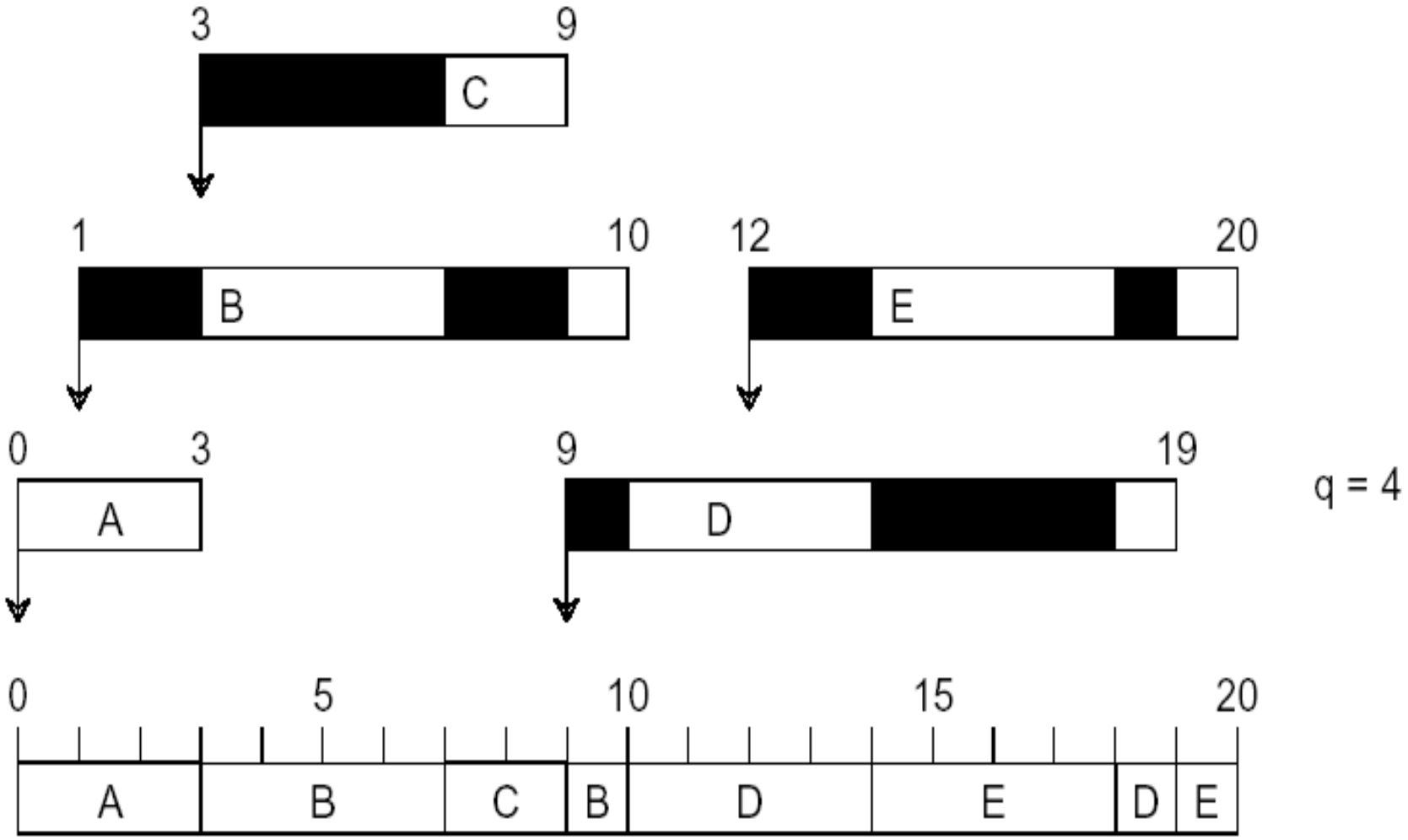


- Typically, higher average turnaround than SJF, but better ***response.***

Another RR Example (q = 1)



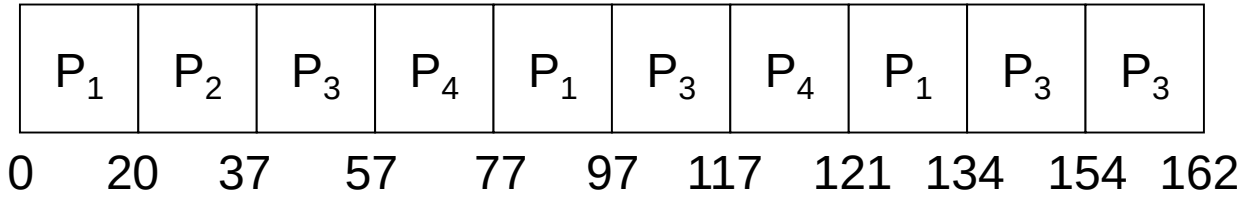
Another RR Example (q = 4)



Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

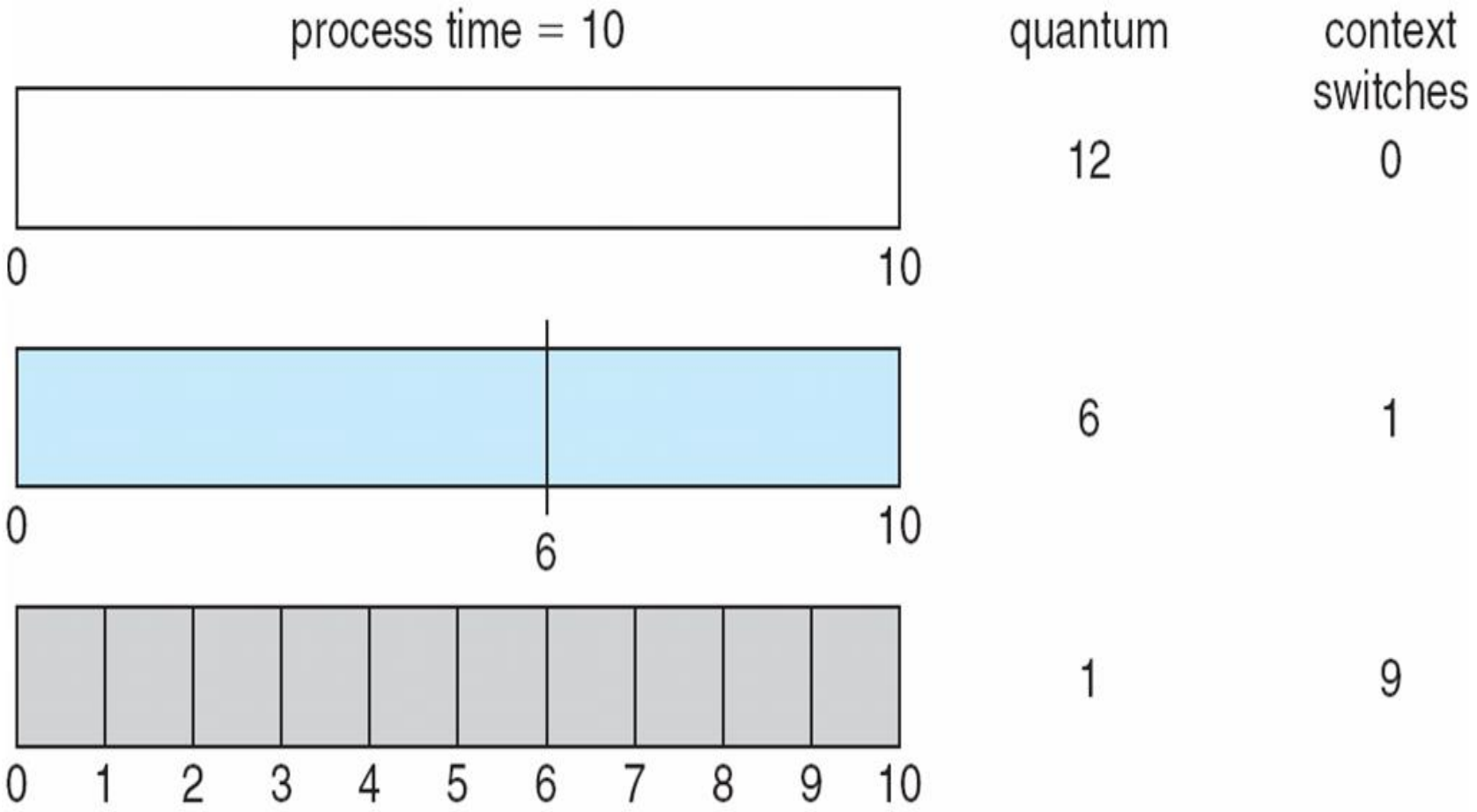


- q should be large compared to context switch time.
- q usually 10ms to 100ms, context switch < 10 usec.

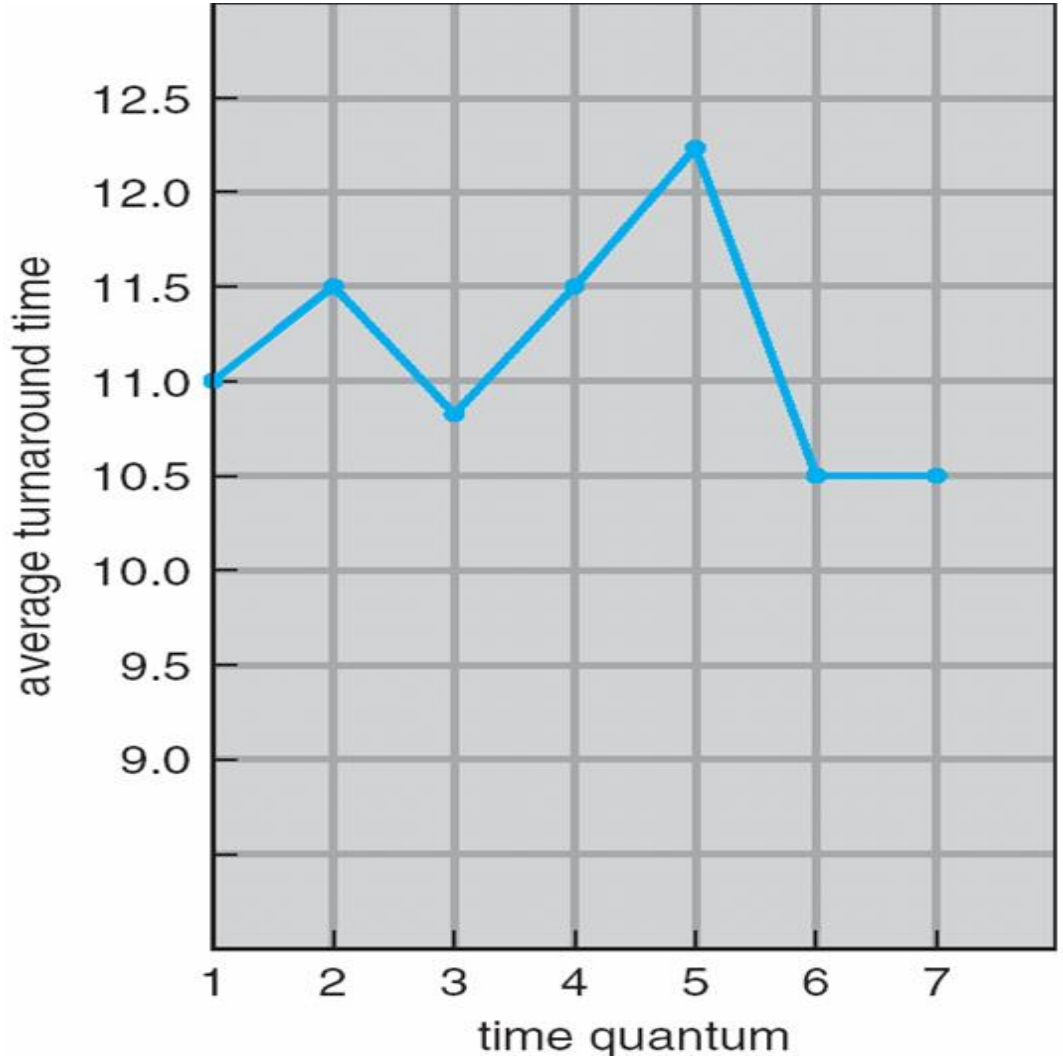
Dynamics of Round Robin (RR)

- Each process gets a *time quantum*, usually 10-100 milliseconds. 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.
- Performance
  - *q* large $\Rightarrow$ FIFO.
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high.

Time Quantum and Context Switch Time



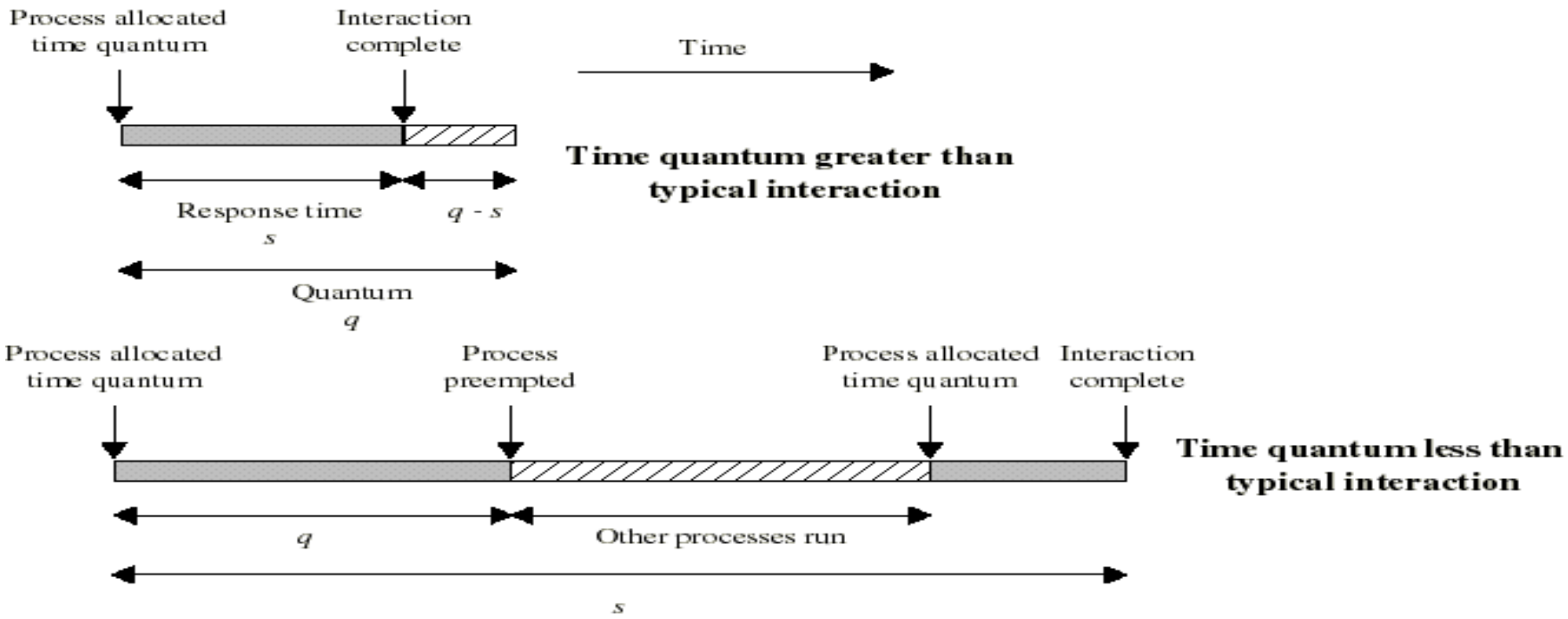
Turnaround Time varies with the Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Time Quantum for Round Robin

- Must be substantially larger than the time required to handle the clock interrupt and dispatching.
- Should be larger then the typical interaction (but not much more to avoid penalizing I/O bound processes).



SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Round Robin Drawbacks

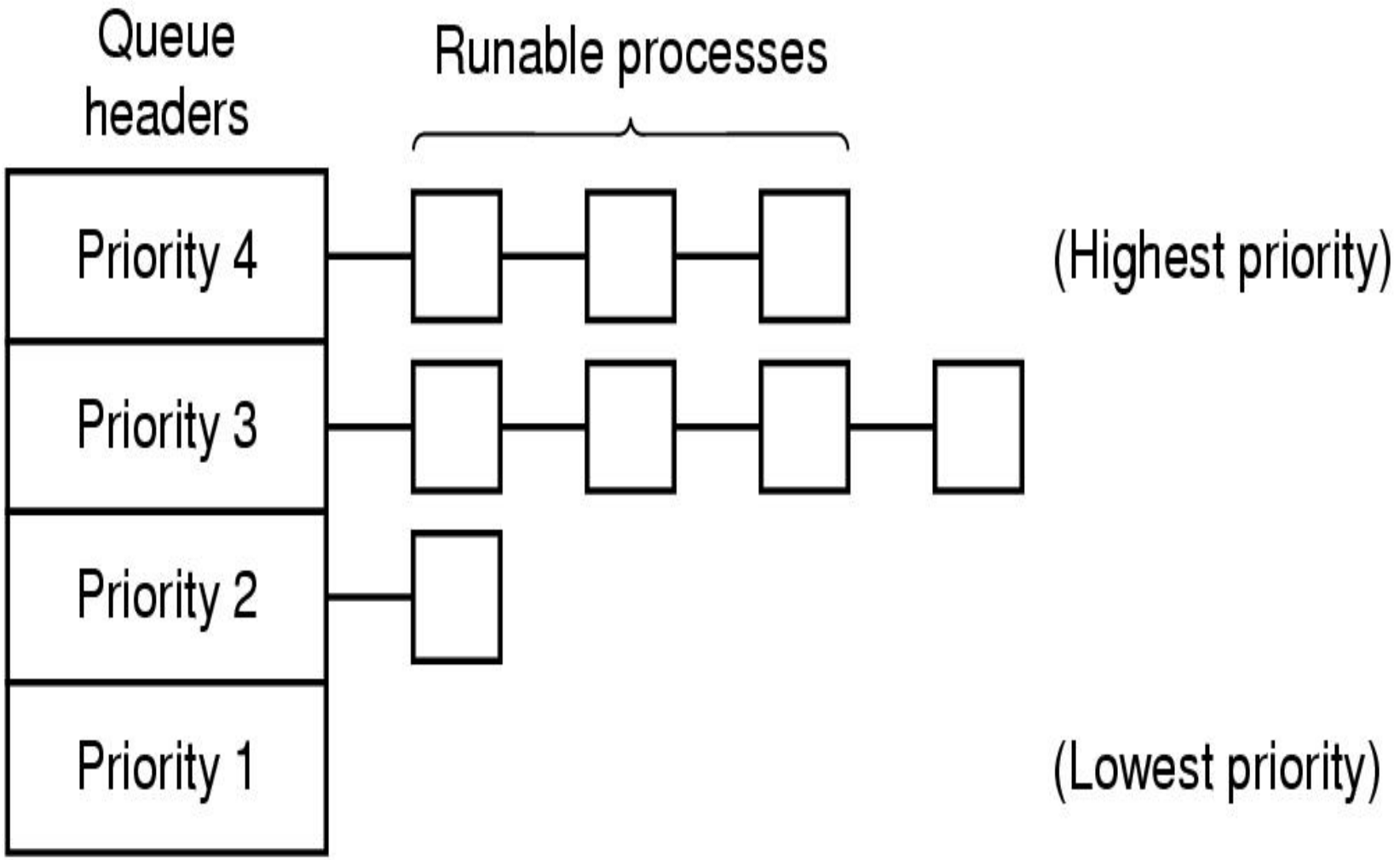
- Still favors CPU-bound processes:
  - A I/O bound process uses the CPU for a time that is less than the time quantum and then blocked waiting for I/O.
  - A CPU-bound process runs for all its time slice and is put back into the ready queue (thus getting in front of blocked processes).
- A solution: virtual round robin:
  - When a I/O has completed, the blocked process is moved to an auxiliary queue which gets preference over the main ready queue.
  - A process dispatched from the auxiliary queue runs no longer than the basic time quantum minus the time spent running since it was selected from the ready queue.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

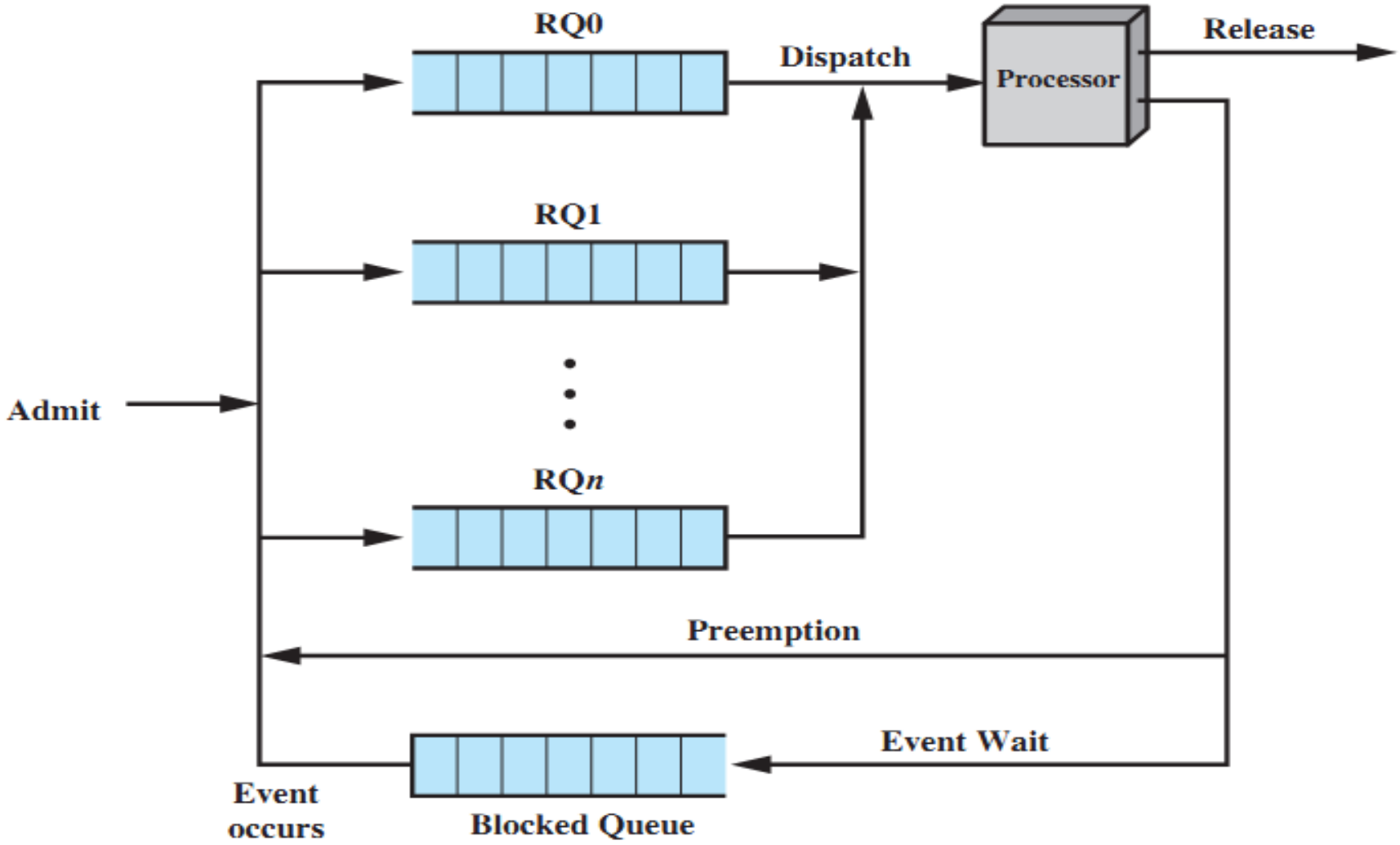
Multiple Priorities

- Implemented by having multiple ready queues to represent each level of priority.
- Scheduler will always choose a process of higher priority over one of lower priority.
- Lower-priority may suffer starvation.
- Then allow a process to change its priority based on its age or execution history.
- Our first scheduling algorithms did not make use of multiple priorities.
- We will now present other algorithms that use dynamic multiple priority mechanisms.

Priority Scheduling with Queues



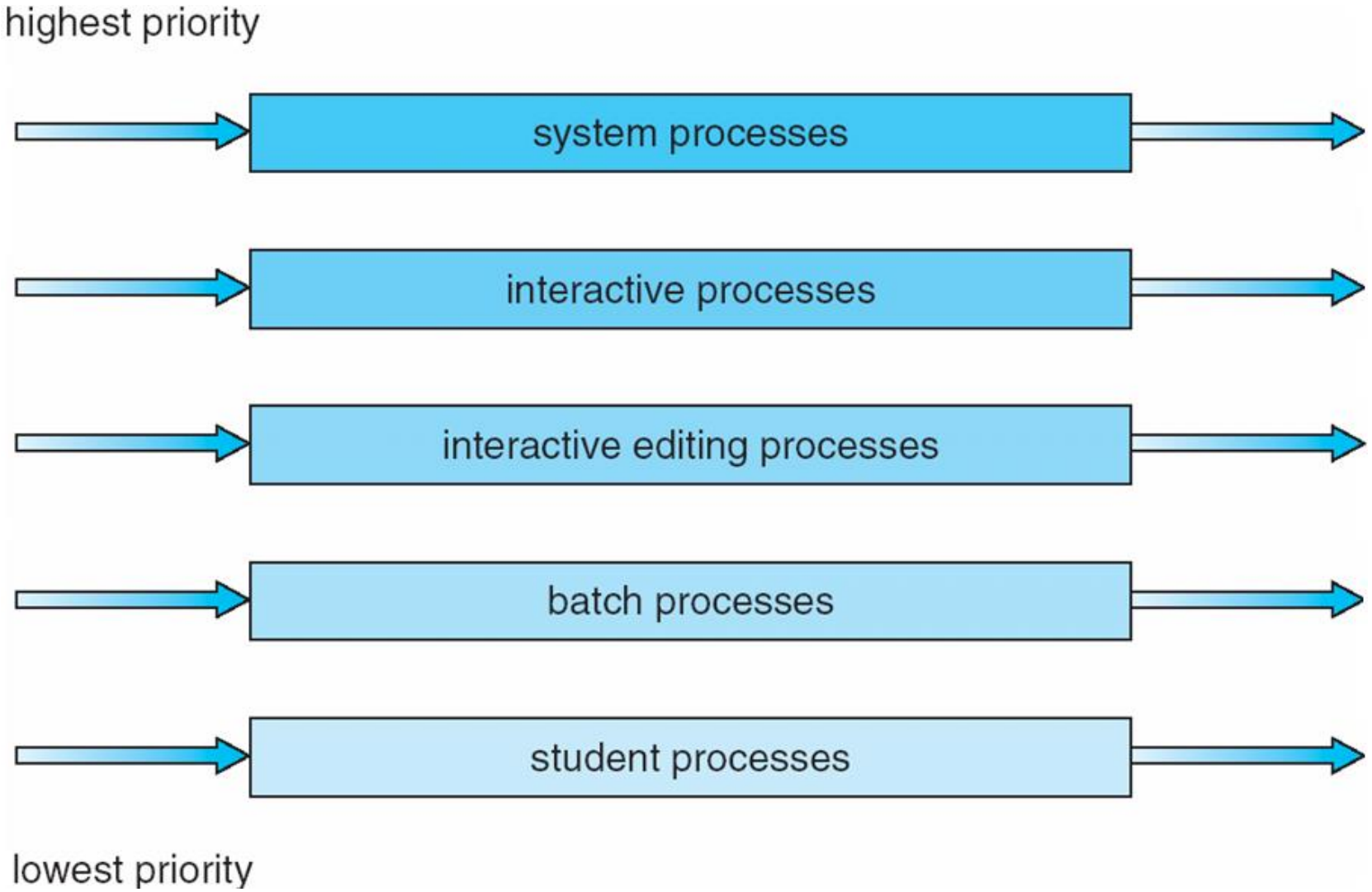
Priority Queuing



Multilevel Queue Scheduling (1)

- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)
- 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 (i.e., serve all from foreground then from background) – 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.

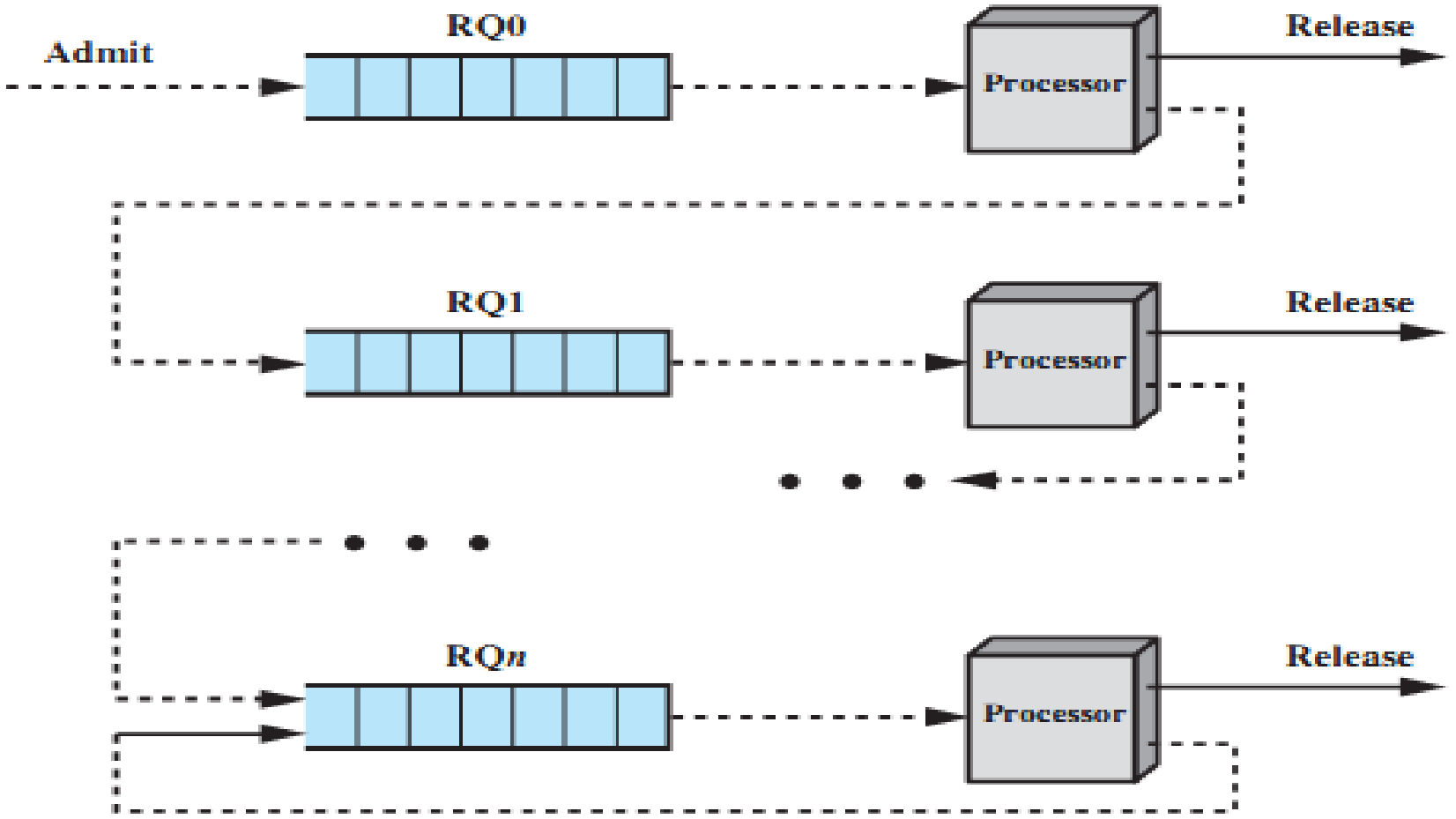
Multilevel Queue Scheduling (2)



Multilevel Feedback Queue

- Preemptive scheduling with dynamic priorities.
- A process can move between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues.
  - scheduling algorithms for each queue.
  - method used to determine which queue a process will enter when that process needs service.
  - method used to determine when to upgrade process.
  - method used to determine when to demote process.

Multiple Feedback Queues



ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

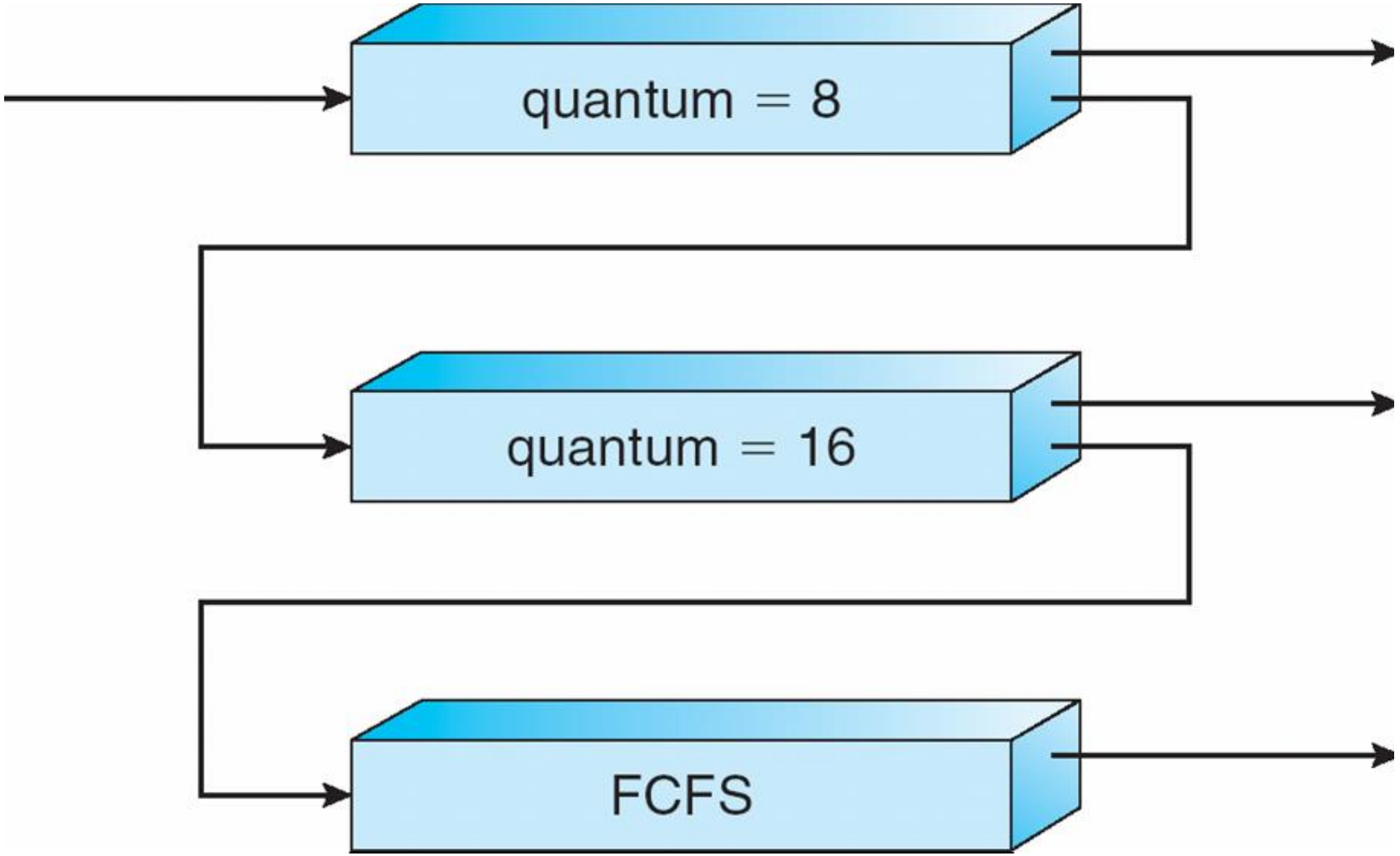
Dynamics of Multilevel Feedback

- Several ready to execute queues with decreasing priorities:
  - P(RQ0) > P(RQ1) > ... > P(RQn).
- New process are placed in RQ0.
- When they reach the time quantum, they are placed in RQ1. If they reach it again, they are place in RQ2... until they reach RQn.
- I/O-bound processes will tend to stay in higher priority queues. CPU-bound jobs will drift downward.
- Dispatcher chooses a process for execution in RQi only if RQi-1 to RQ0 are empty.
- Hence long jobs may starve.

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR with time quantum 16 milliseconds
  - Q_2 – FCFS
- Scheduling:
  - A new job enters queue Q_0 which is served FCFS. When it gets 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 and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.
  - Could be also vice versa.

Multilevel Feedback Queues

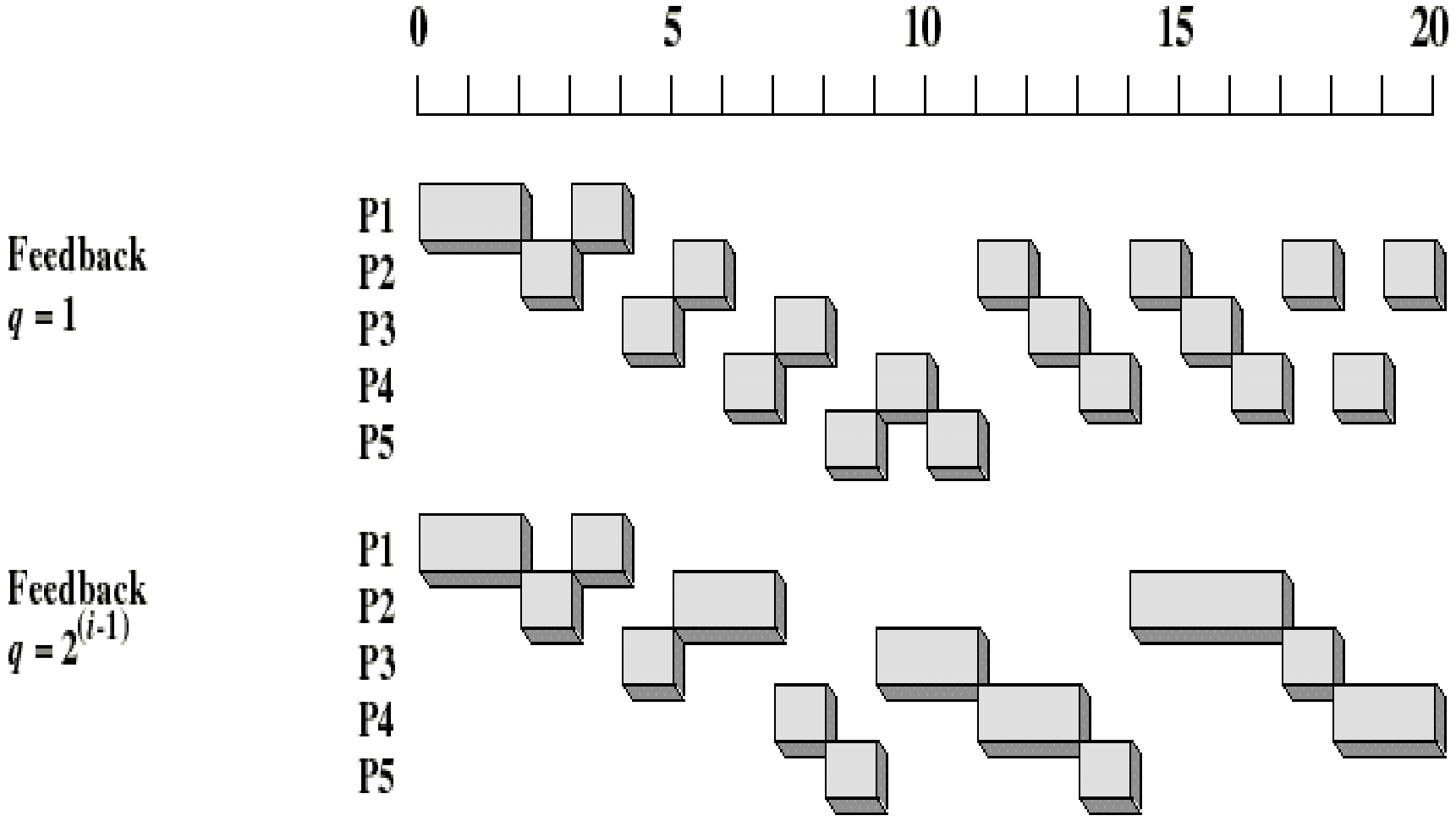


ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Time Quantum for Feedback Scheduling

- With a fixed quantum time, the turnaround time of longer processes can stretch out alarmingly.
- To compensate we can increase the time quantum according to the depth of the queue:
  - Example: time quantum of RQi = 2^{i-1}
- See next slide for an example.
- Longer processes may still suffer starvation. Possible fix: promote a process to higher priority after some time.

Time Quantum for Feedback Scheduling



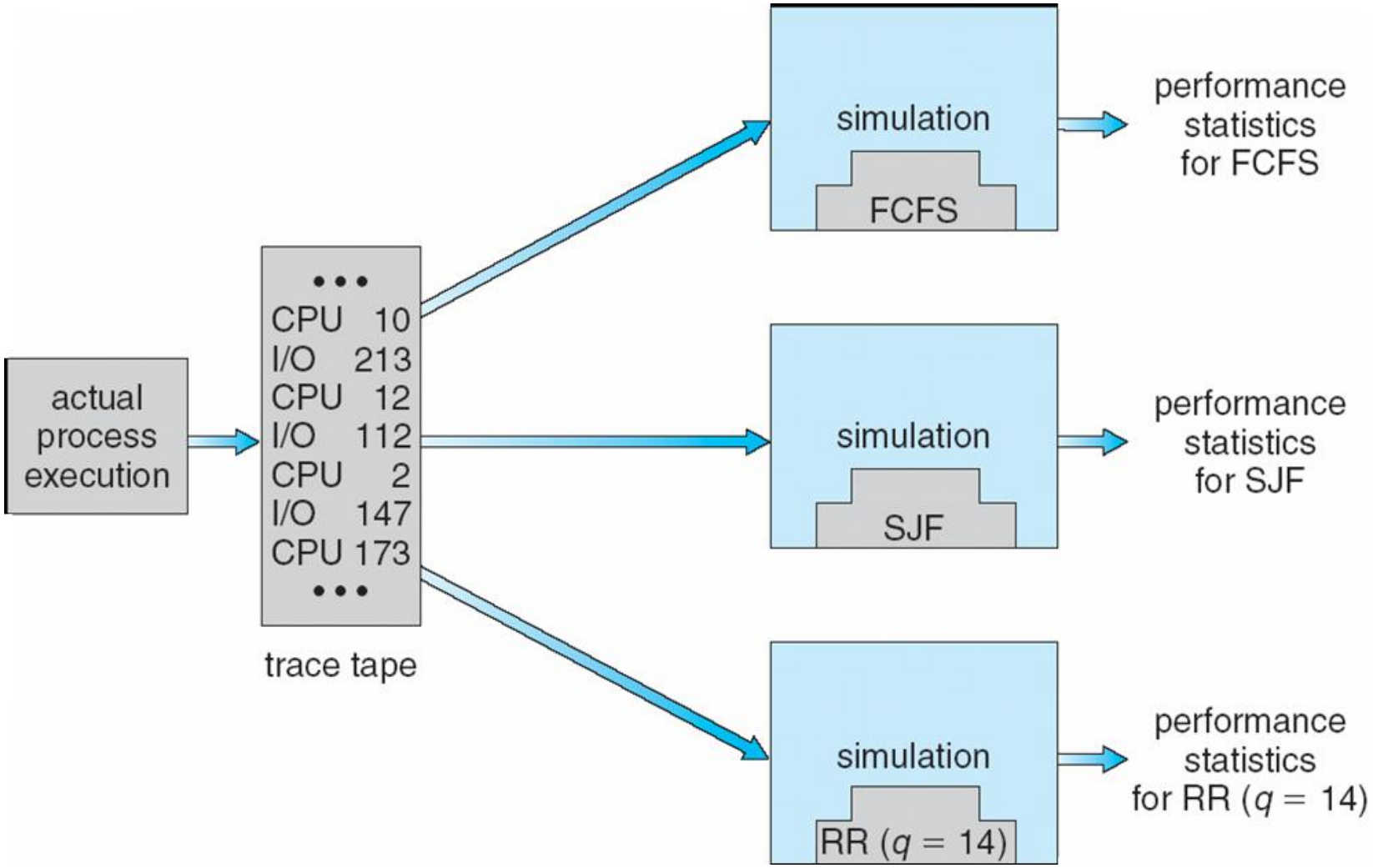
Algorithms Comparison

- Which one is best?
- The answer depends on:
  - on the system workload (extremely variable).
  - hardware support for the dispatcher.
  - relative weighting of performance criteria (response time, CPU utilization, throughput...).
  - The evaluation method used (each has its limitations...).
- Hence the answer depends on too many factors to give any...

Scheduling Algorithm Evaluation

- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- Queuing models
- Simulations
- Implementation

Evaluation of CPU Schedulers by Simulation

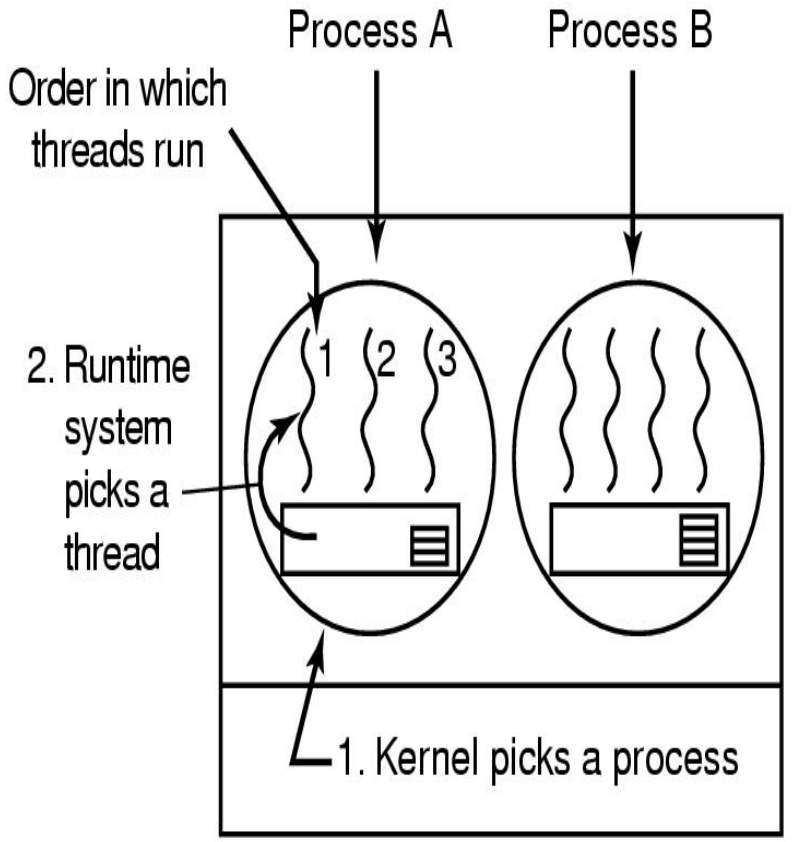


ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Thread Scheduling

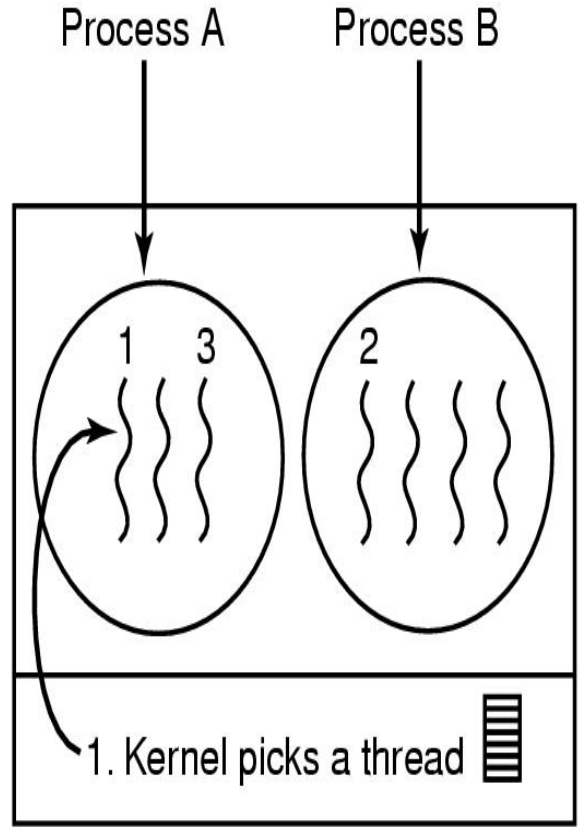
- Depends if ULT or KLT or mixed.
- Local Scheduling – How the threads library decides which ready thread to run.
- ULT can employ an application-specific thread scheduler.
- Global Scheduling – How the kernel decides which kernel thread to run next.
- KLT can employ priorities within thread scheduler.

Thread Scheduling Example



Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3

(a)

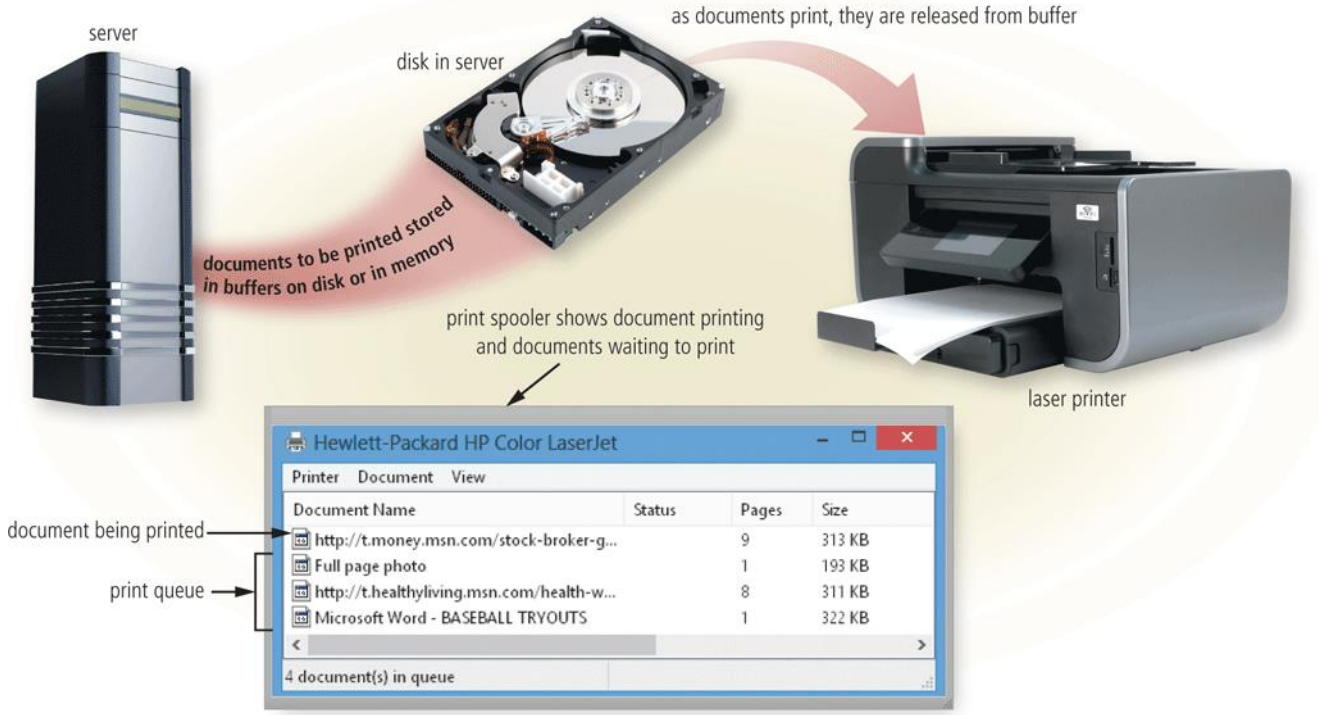


Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3

(b)

Operating System Functions

- The operating system determines the order in which tasks are processed



Operating System Functions

- Operating systems often provide users with a variety of tools related to managing a computer, its devices, or its programs

File Manager	Search	Image Viewer	Uninstaller	Disk Cleanup
Disk Defragmenter	Screen Saver	File Compression	PC Maintenance	Backup and Restore

ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

FileSystem Interface

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Contents

- File Concept
- Access Methods
- Directory Structures
- File System Mounting
- File Sharing
- Protection

File Concept

- Contiguous logical address space.
- Types:
  - Data
    - numeric
    - character
    - binary
  - Program
- Contents defined by file's creator:
  - Many types:
    - Consider text file, source file, executable file

File Structure

- None – sequence of words, bytes.
- Simple record structure:
  - Lines
  - Fixed length
  - Variable length
- Complex Structures:
  - Formatted document
  - Relocatable load file
- Can simulate last two with first method by inserting appropriate control characters.
- Who decides:
  - Operating system
  - Program

Basic file attributes

- **Name** – only information kept in human-readable form.
- **Identifier** – unique tag (number) identifies file within system.
- **Type** – needed for systems that support different types.
- **Location** – pointer to file location on device.
- **Size** – current file size.
- **Protection** – controls who can do reading, writing, executing.
- **Time, date, and user identification** – data for protection, security, and usage monitoring.
- Information about files are kept in the directory structure, which is maintained on the disk.
- Many variations, including extended file attributes such as file checksum.
- Information kept in the directory structure.

More file attributes

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

File Type Extensions (1)

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

File Type Extensions (2)

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes com-pressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Common File Operations

- Create
- Read/Write – at read/write pointer location.
- Seek: Reposition within file.
- Delete
- Append/Truncate
- Set/Get Attributes
- Open(F_i): search the directory structure on disk for entry F_i, and move the content of entry to memory.
- Close (F_i): move the content of entry F_i in memory to directory structure on disk.

Open Files

- Several pieces of data needed to manage open files:
  - Open-file table: tracks open files.
  - File pointer: pointer to last read/write location, per process that has the file open.
  - File-open count: counter of number of times a file is open – to allow removal of data from open-file table when last processes closes it.
  - Disk location of the file: cache of data access information.
  - Access rights: per-process access mode information.

Access Methods

- **Sequential Access**

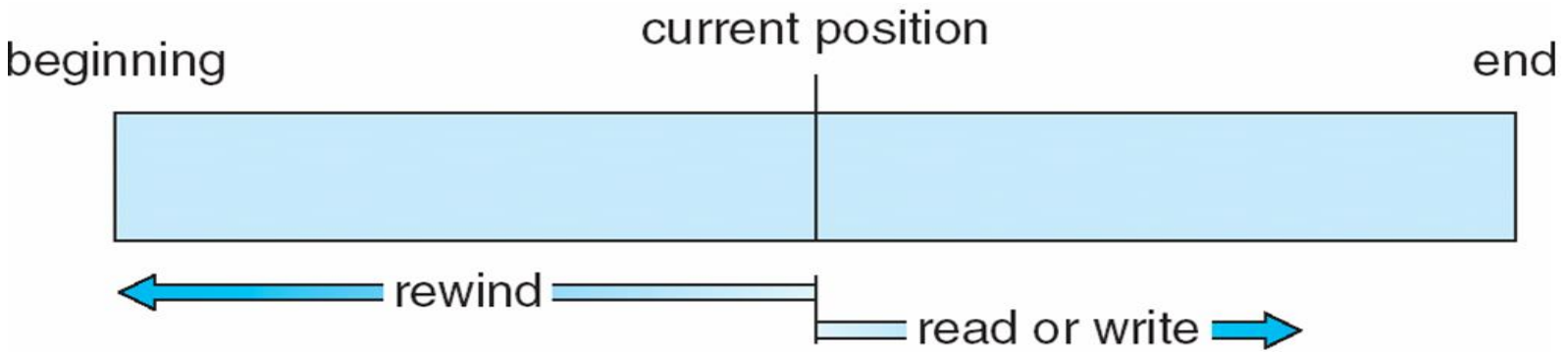
  read next
  write next
  reset
  no *read after last write*
  (rewrite)

- **Direct Access –** file is fixed length logical records

  read n
  write n
  position to n
  read next
  write next
  rewrite n

  n = relative block number

Sequential/Direct Access

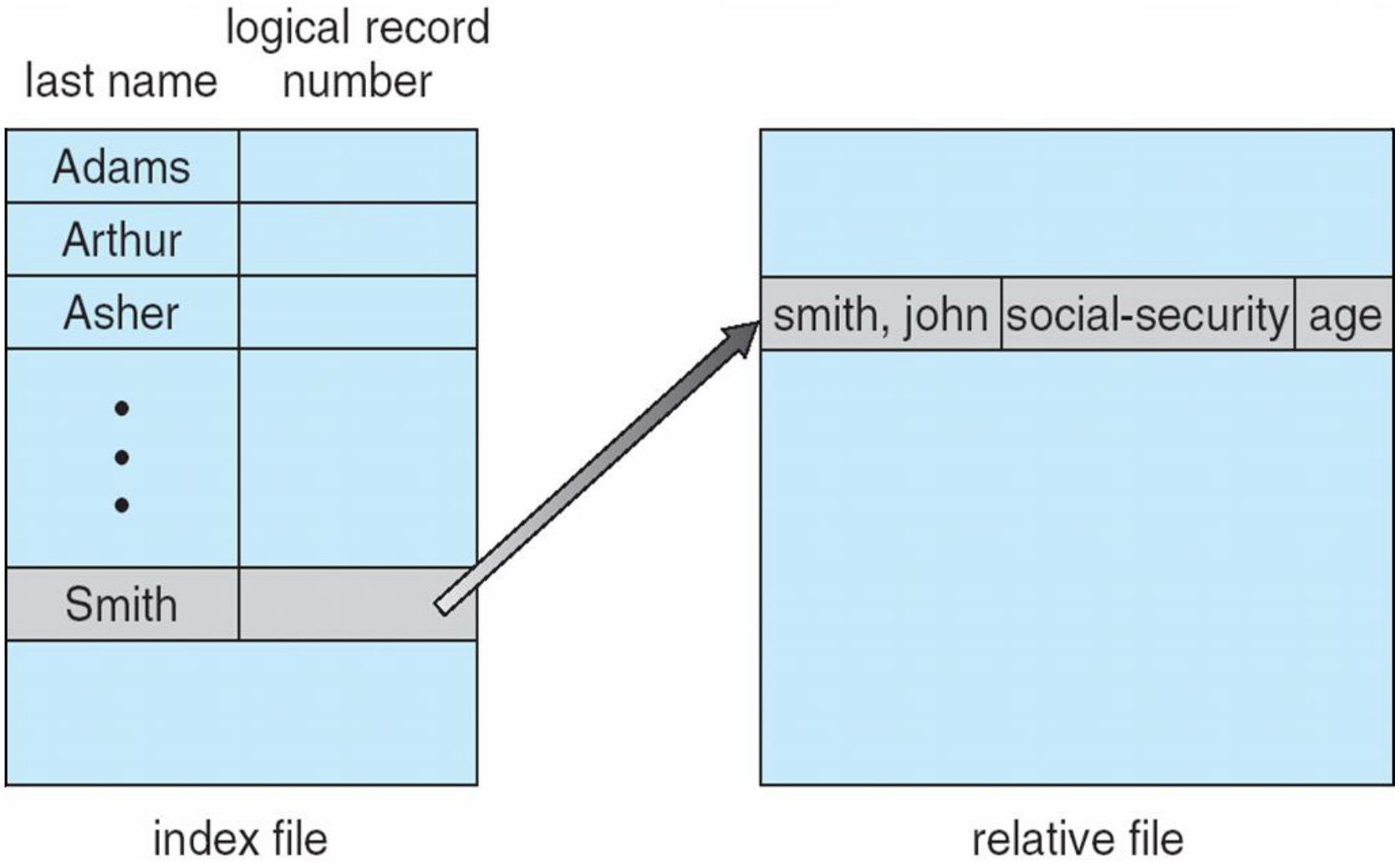


sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

Other Access Methods

- Can be built on top of base methods.
- General involve creation of an index for the file.
- Keep index in memory for fast determination of location of data to be operated on (consider UPC code plus record of data about that item).
- If too large, index (in memory) of the index (on disk).
- IBM indexed sequential-access method (ISAM):
  - Small master index, points to disk blocks of secondary index
  - File kept sorted on a defined key
  - All done by the OS.
- VMS operating system provides index and relative files as another example (see next slide).

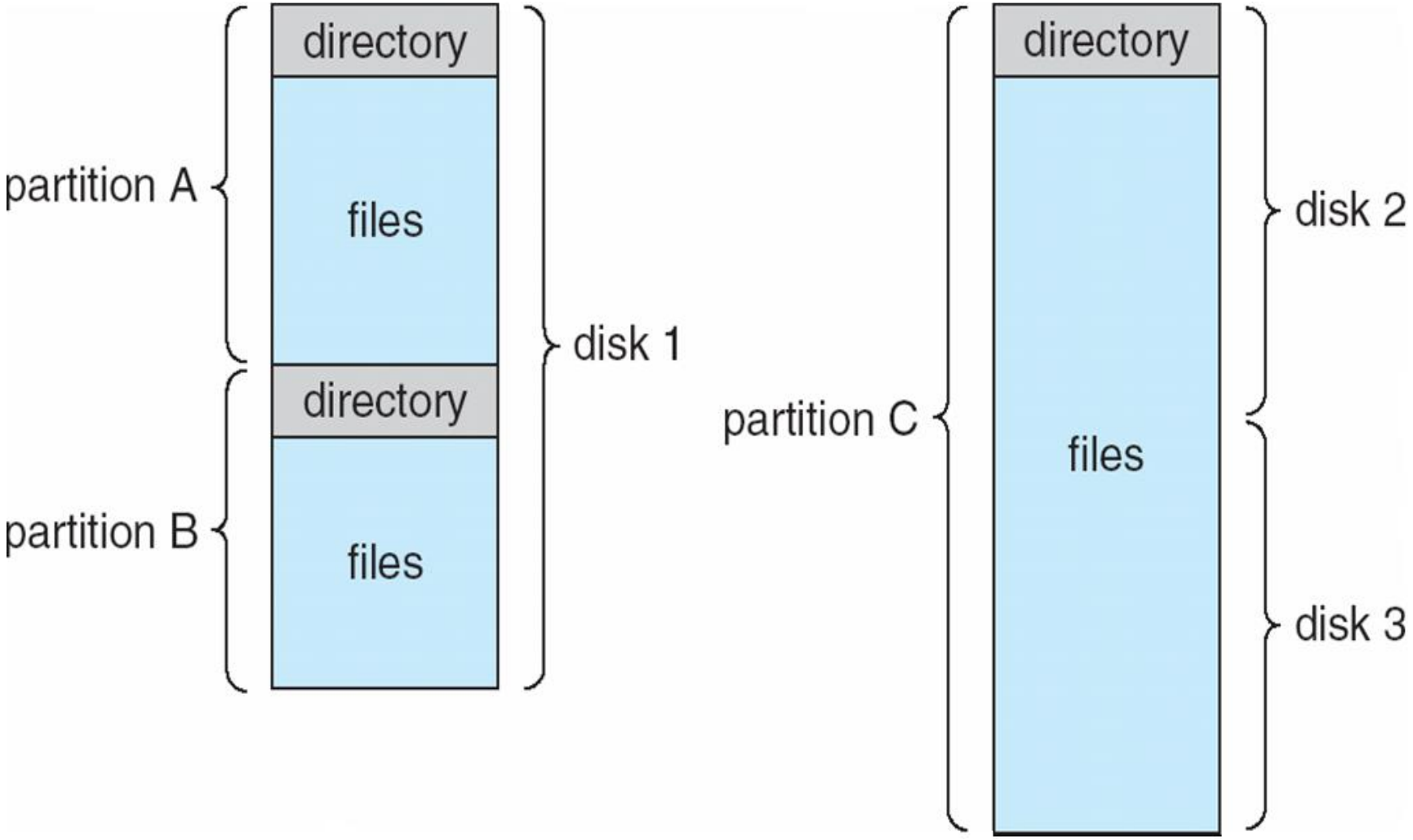
Example of Index and Relative Files



Disk Structure

- Disk can be subdivided into partitions.
- Disks or partitions can be RAID protected against failure.
- Disk or partition can be used raw – without a file system, or formatted with a file system.
- Partitions also known as minidisks, slices.
- Entity containing file system known as a volume.
- Each volume containing file system also tracks that file system's info in device directory or volume table of contents.
- As well as general-purpose file systems, there are many special-purpose file systems, frequently all within the same operating system or computer.

A Typical File-system Organization



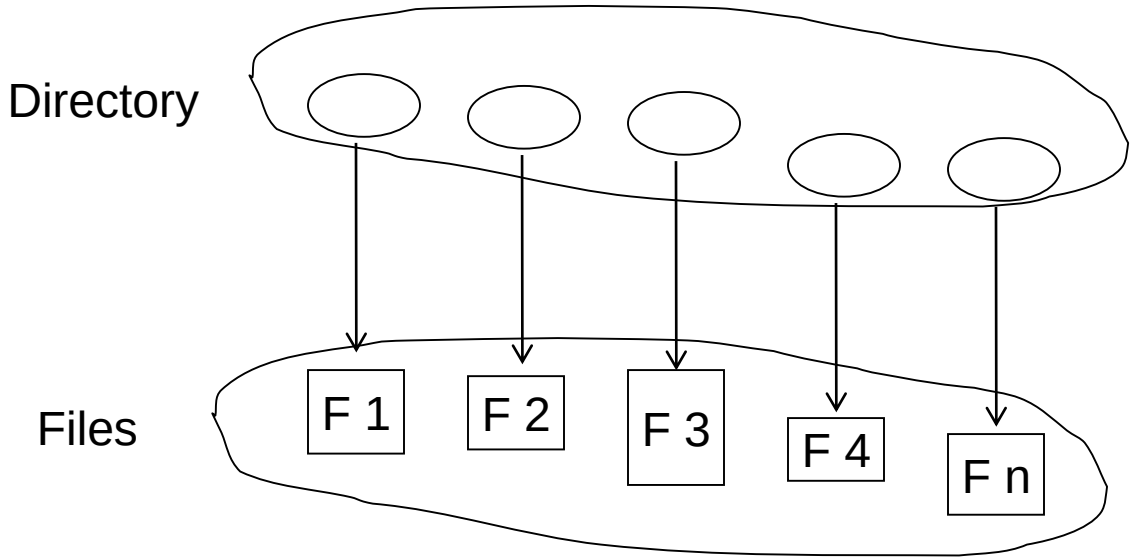
ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Types of File Systems

- We mostly talk of general-purpose file systems.
- But systems frequently have may file systems, some general- and some special- purpose.
- Consider Solaris has:
  - tmpfs – memory-based volatile FS for fast, temporary I/O
  - objfs – interface into kernel memory to get kernel symbols for debugging
  - ctfs – contract file system for managing daemons
  - lofs – loopback file system allows one FS to be accessed in place of another
  - procfs – kernel interface to process structures
  - ufs, zfs – general purpose file systems

Directory Structures

- Collection of nodes containing information about all files.



Both the directory structure and the files reside on disk.
Backups of these two structures are kept on tapes.

Information in a Directory Entry

- Name
- Type
- Address
- Current length
- Maximum length
- Date last accessed (for archival)
- Date last updated (for dump)
- Owner ID (who pays)
- Protection information

Directory Operations

- Operations performed on a directory:
  - Search for a file
  - Create a file
  - Delete a file
  - List a directory
  - Rename a file
  - Traverse the file system

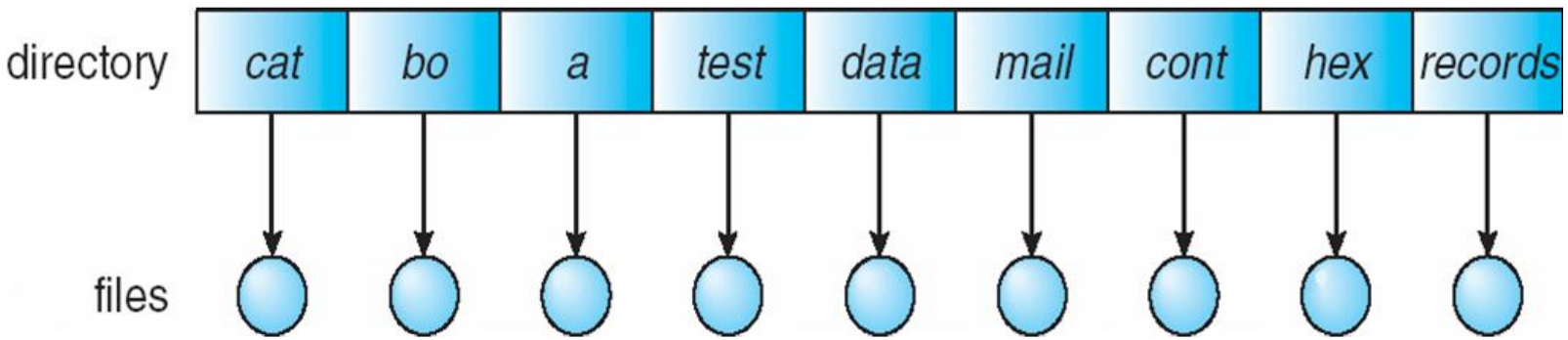
Directory Organization Characteristics

Organize the directory (logically) to obtain:

- **Efficiency** – locating a file quickly.
- **Naming** – convenient to users:
  - Two users can have same name for different files.
  - The same file can have several different names.
- **Grouping** – logical grouping of files by properties, (e.g., all Java programs,
  all games, …).

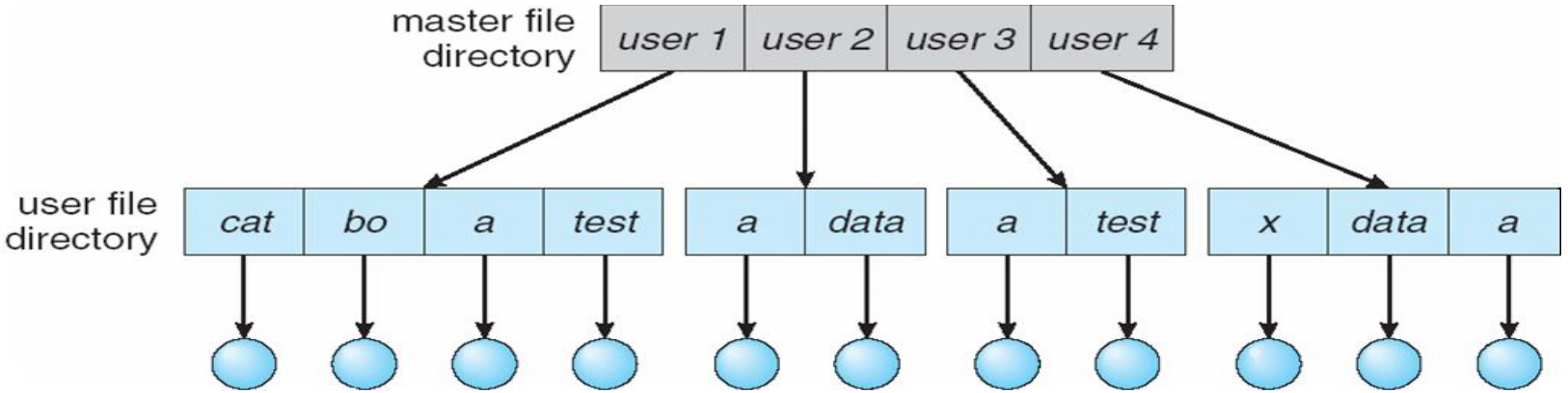
Single-Level Directory

- A single directory for all users.
- Common problems:
  - Eventual length of directory.
  - Giving unique names to files.
  - Remembering names of files.
  - Grouping of files (use file extensions).

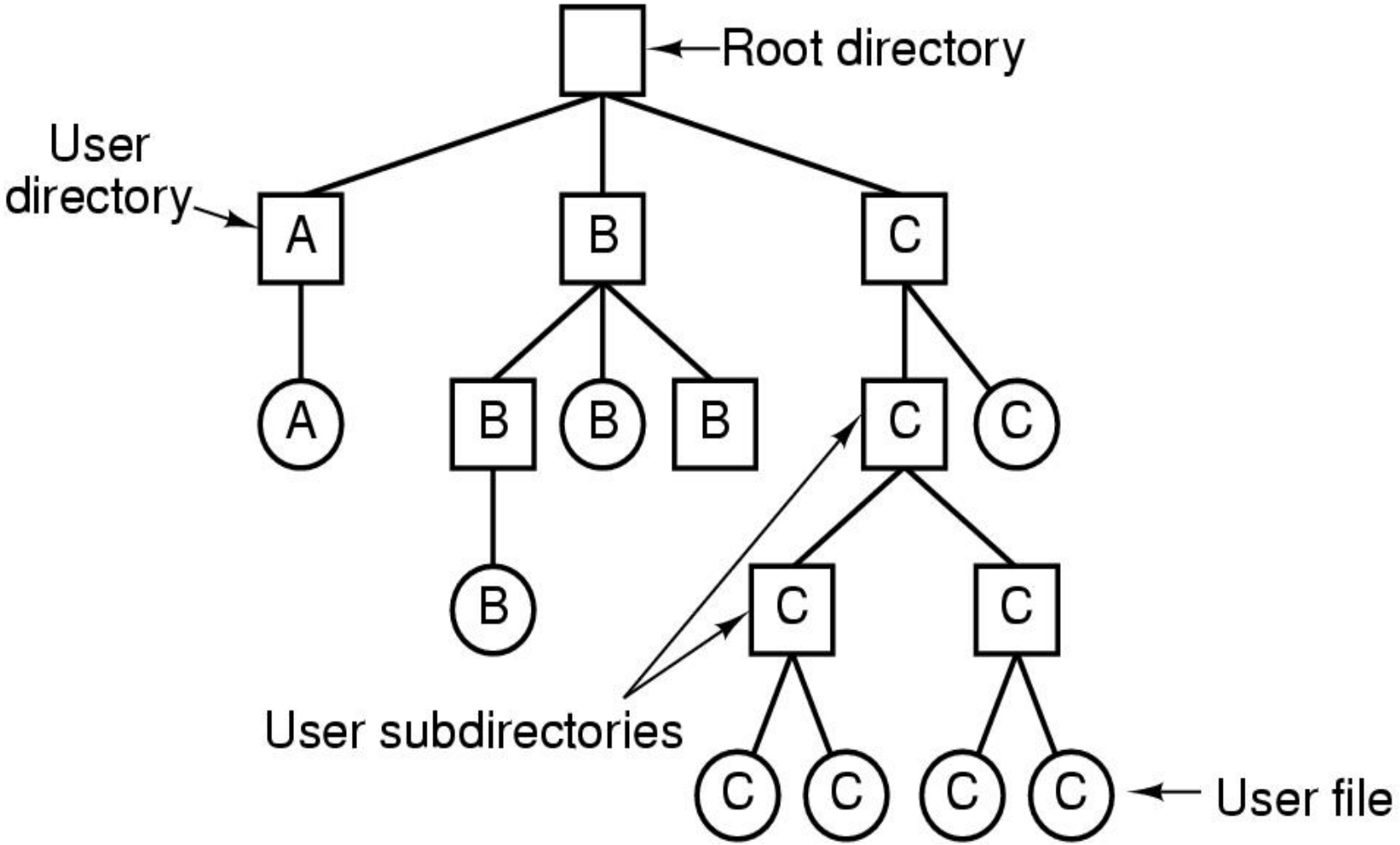


Two-Level Directory

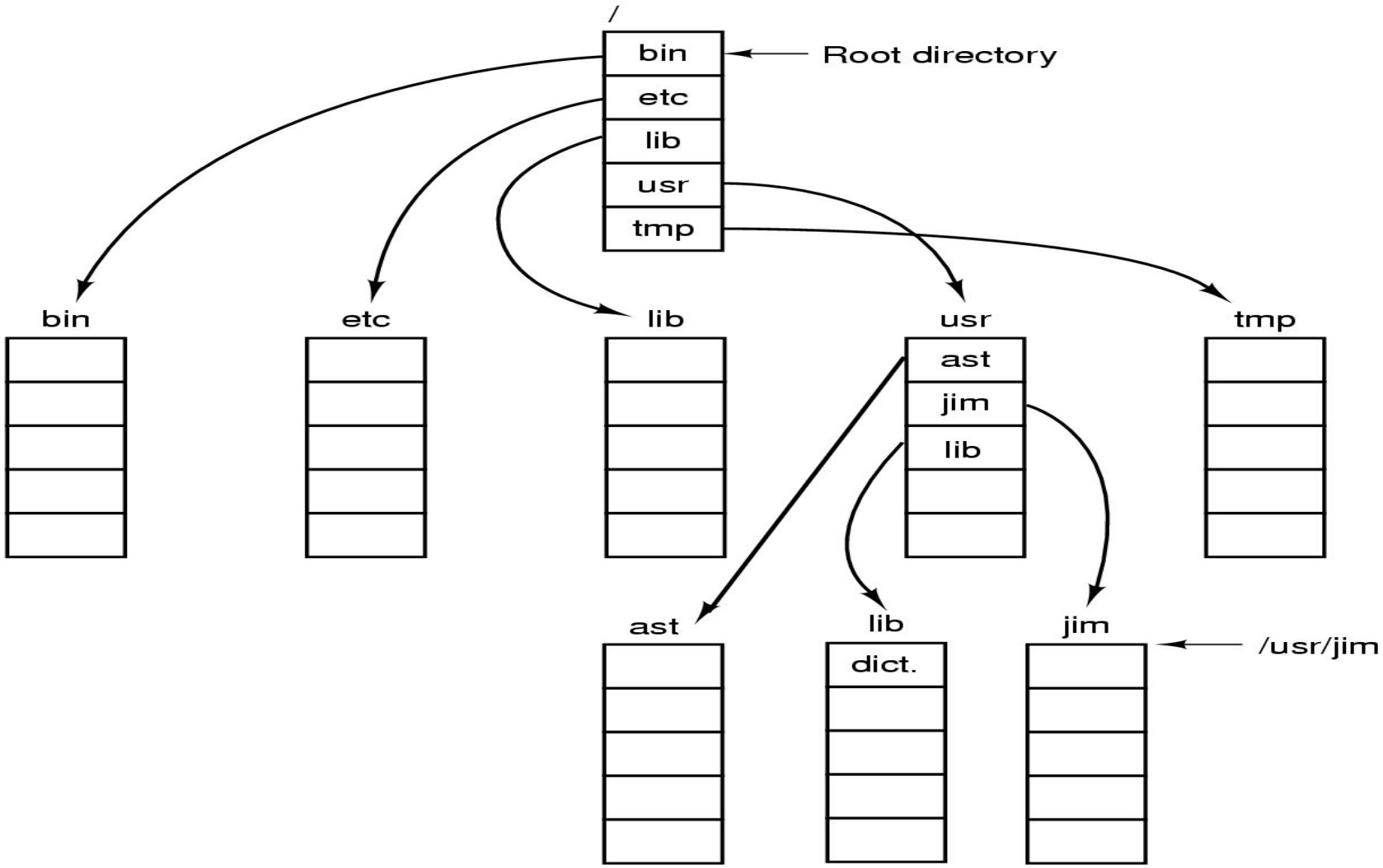
- Separate directory for each user.
- Use of path name.
- Can have the same file name for different users.
- Provides efficient searching.
- No grouping capability.
- Main problem: violates zero-one-infinity principle.



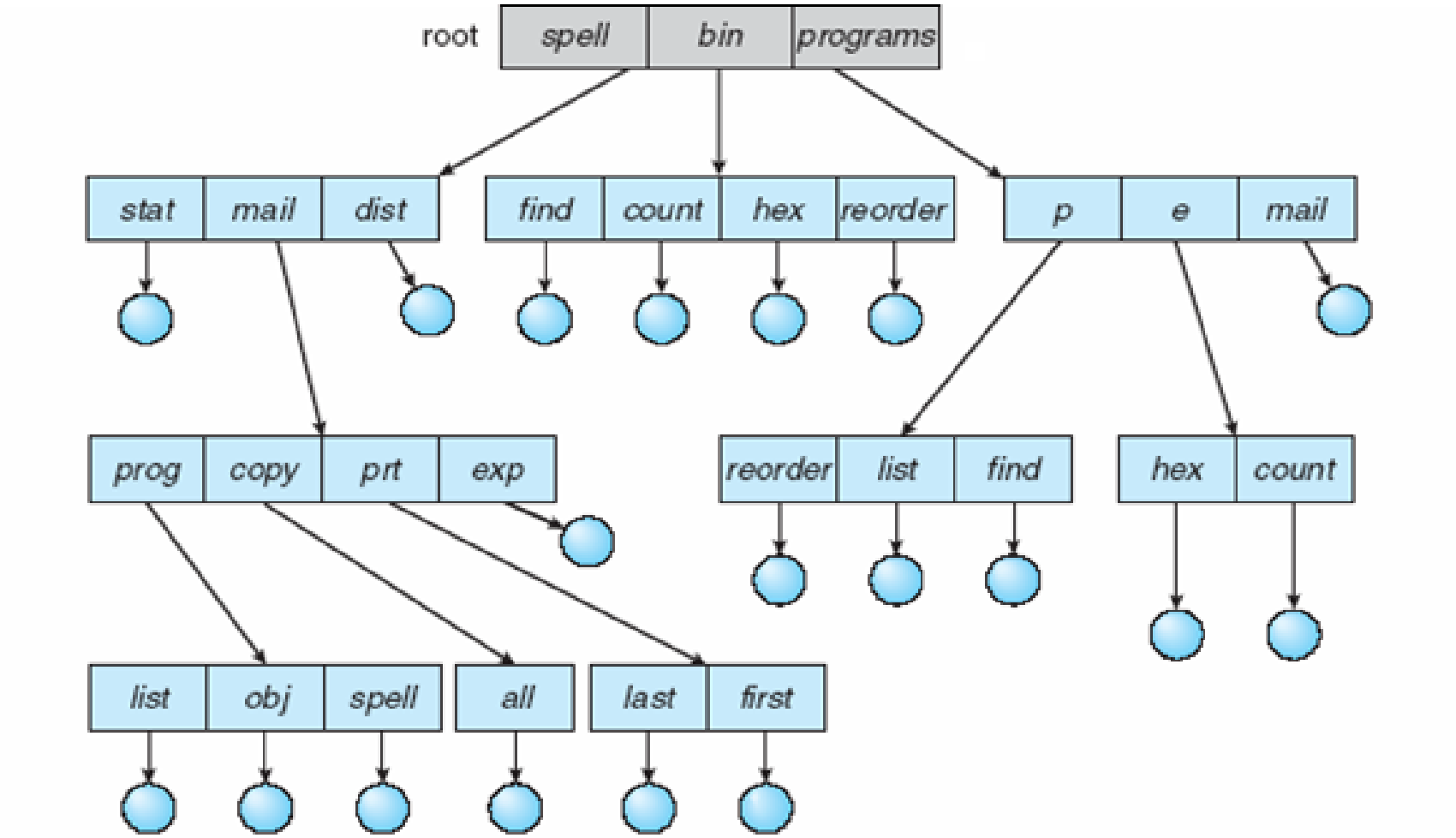
Tree-Structured Directory Components



Directory Path Names



Tree-Structured Directory



Tree-Structured Directories (1)

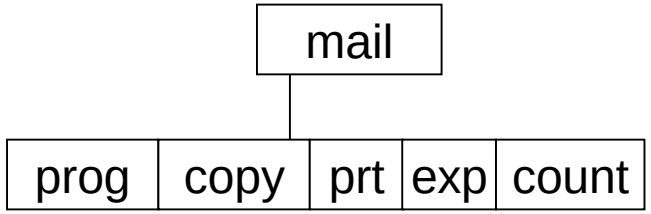
- Provides efficient searching.
- Has grouping capability.
- Current directory (working directory):
  - **cd** /spell/mail/prog
  - **type** list
- Use absolute or relative path name.

Tree-Structured Directories (2)

- Creating a new file is done in current directory.
- Delete a file: **rm** <file-name>
- Creating a new subdirectory is done in current directory: **mkdir** <dir-name>

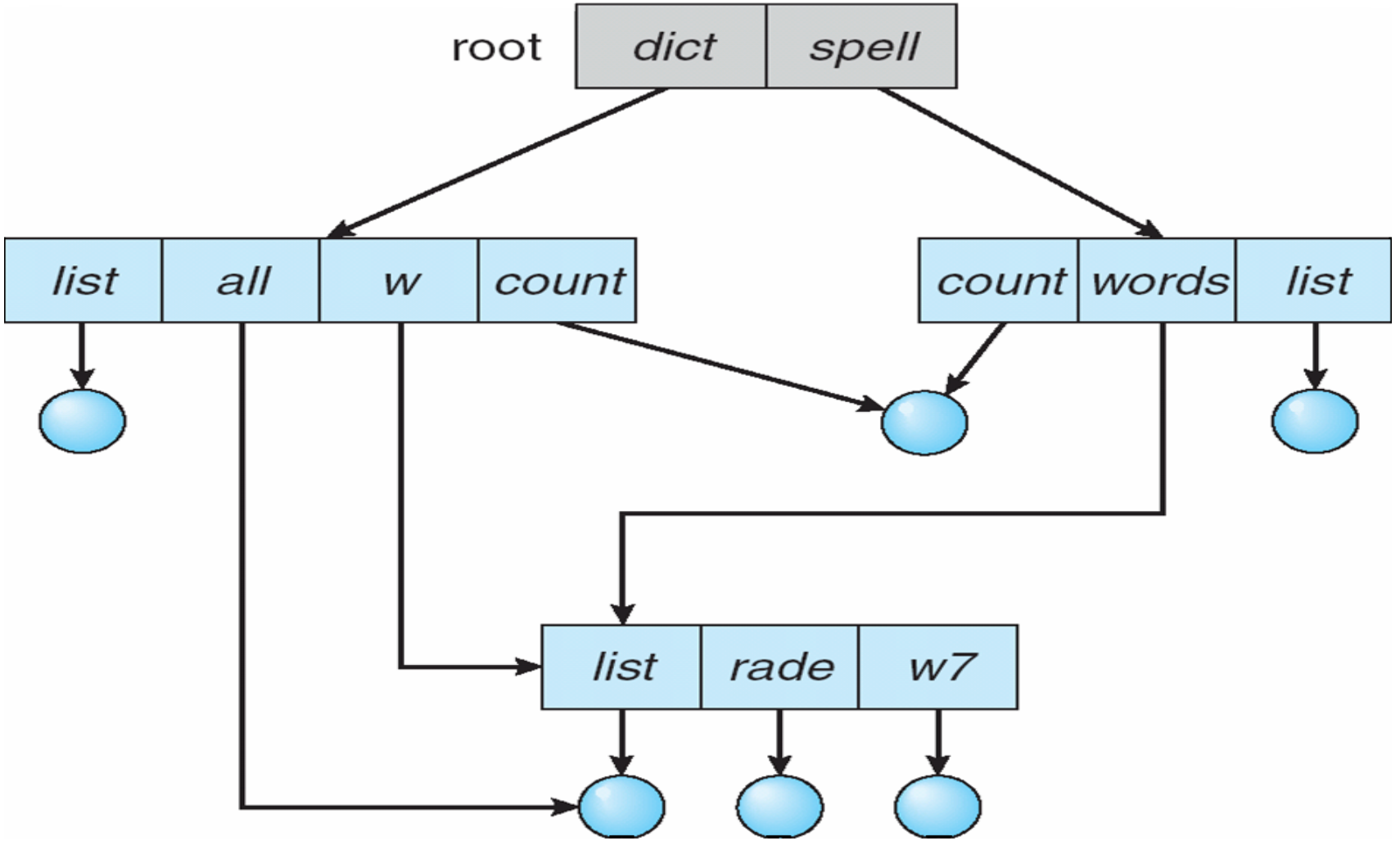
  Example: if in current directory **/mail**

  mkdir count



- Deleting “mail” ⇒ deleting the entire subtree rooted by “mail”.

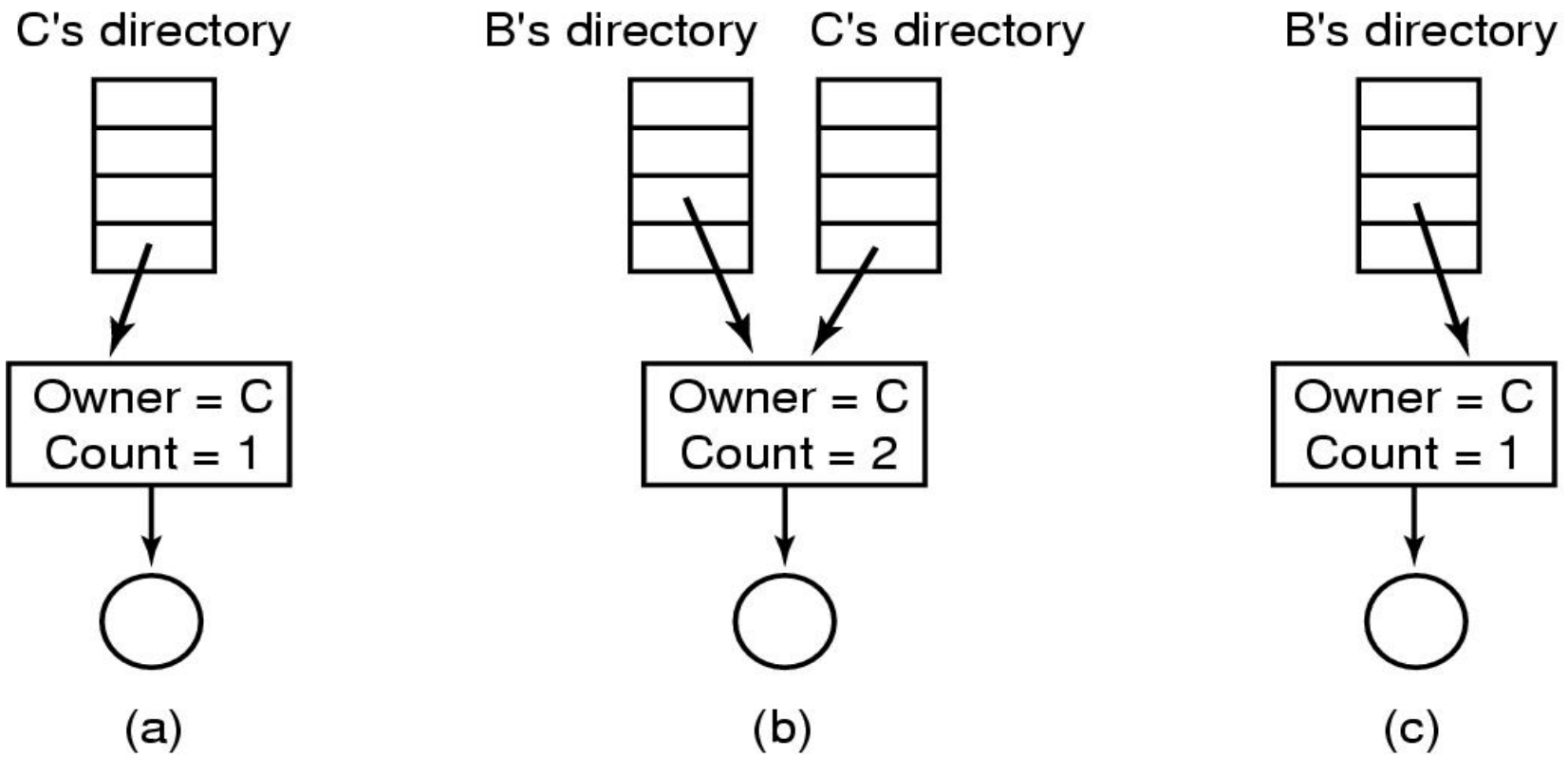
Directed Acyclic Graph (DAG) Directories



Directed Acyclic Graph Directories

- Have shared subdirectories and files.
- Entry may have two different names (aliasing).
- If *dict* deletes *list* $\Rightarrow$ dangling pointer – solutions:
  - Backpointers, so we can delete all pointers.
    Variable size records a problem.
  - Backpointers using a daisy chain organization.
  - Entry-hold-count solution.
- Newer type of directory entry:
  - **Link** – another name (pointer) to an existing file
  - **Resolve the link** – follow pointer to locate the file

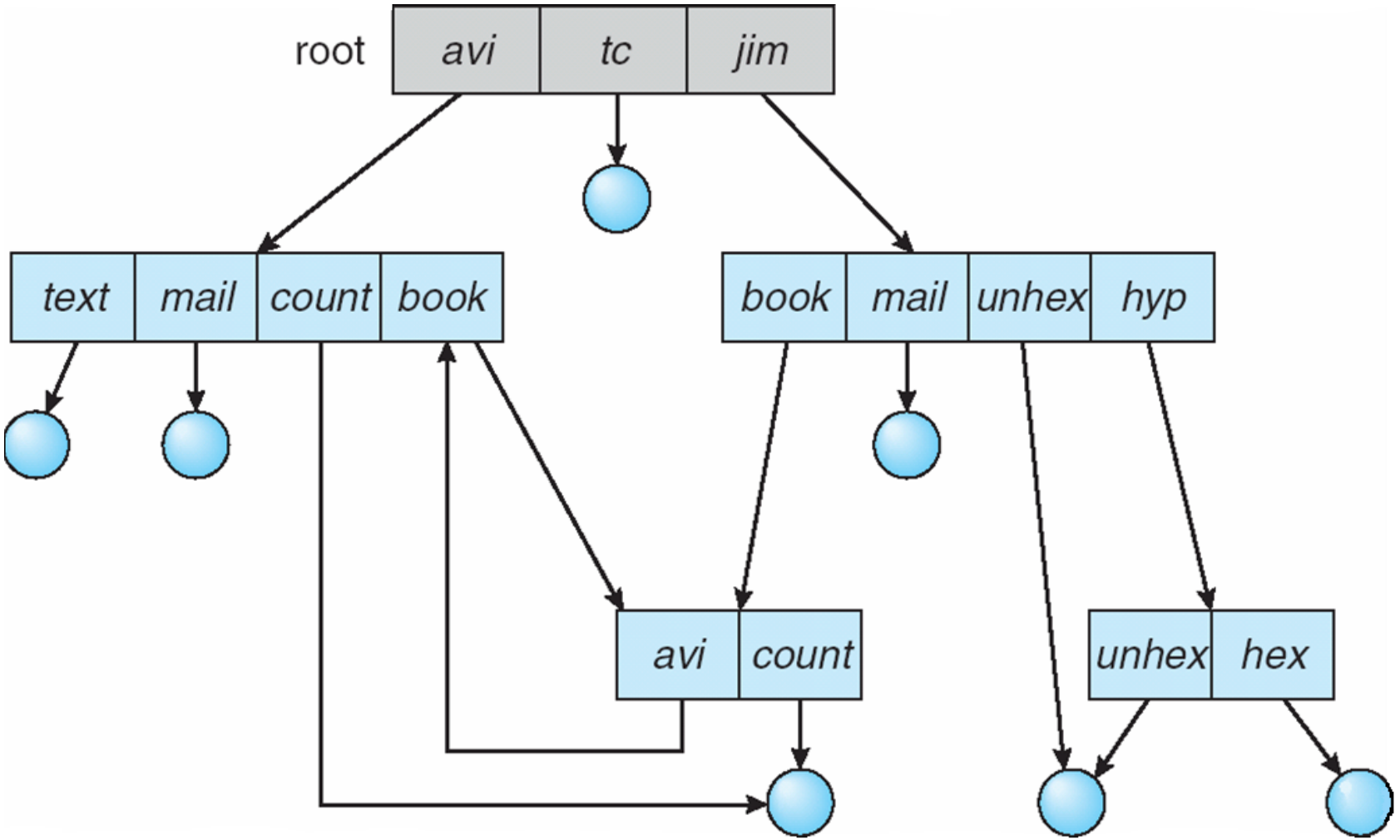
Shared File Example



(a) Situation prior to linking. (b) After the link is created.

(c) After the original owner removes the file.

General Graph Directory



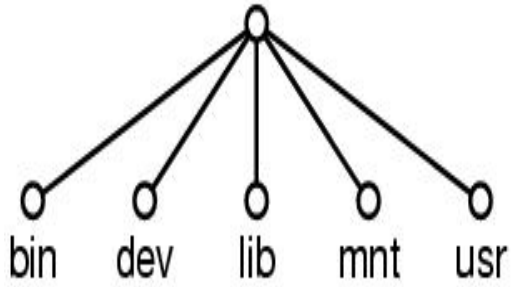
ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

General Graph Directory

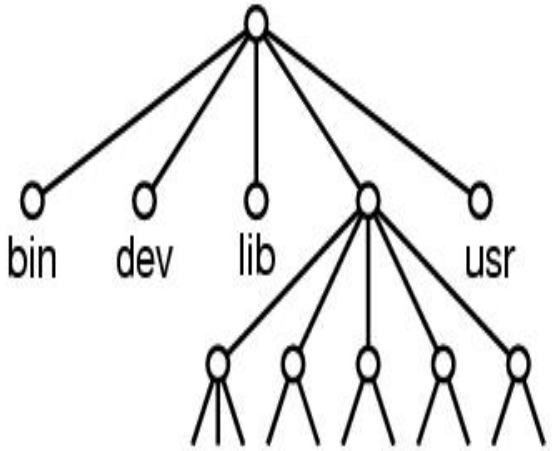
- How do we guarantee no cycles?
  - Allow only links to files, not subdirectories.
  - Every time a new link is added use a cycle detection algorithm to determine whether it is OK.
  - Use garbage collection.

File System Mounting

- A file system must be **mounted** before it can be accessed.
- A unmounted file system is mounted at a **mount point.**

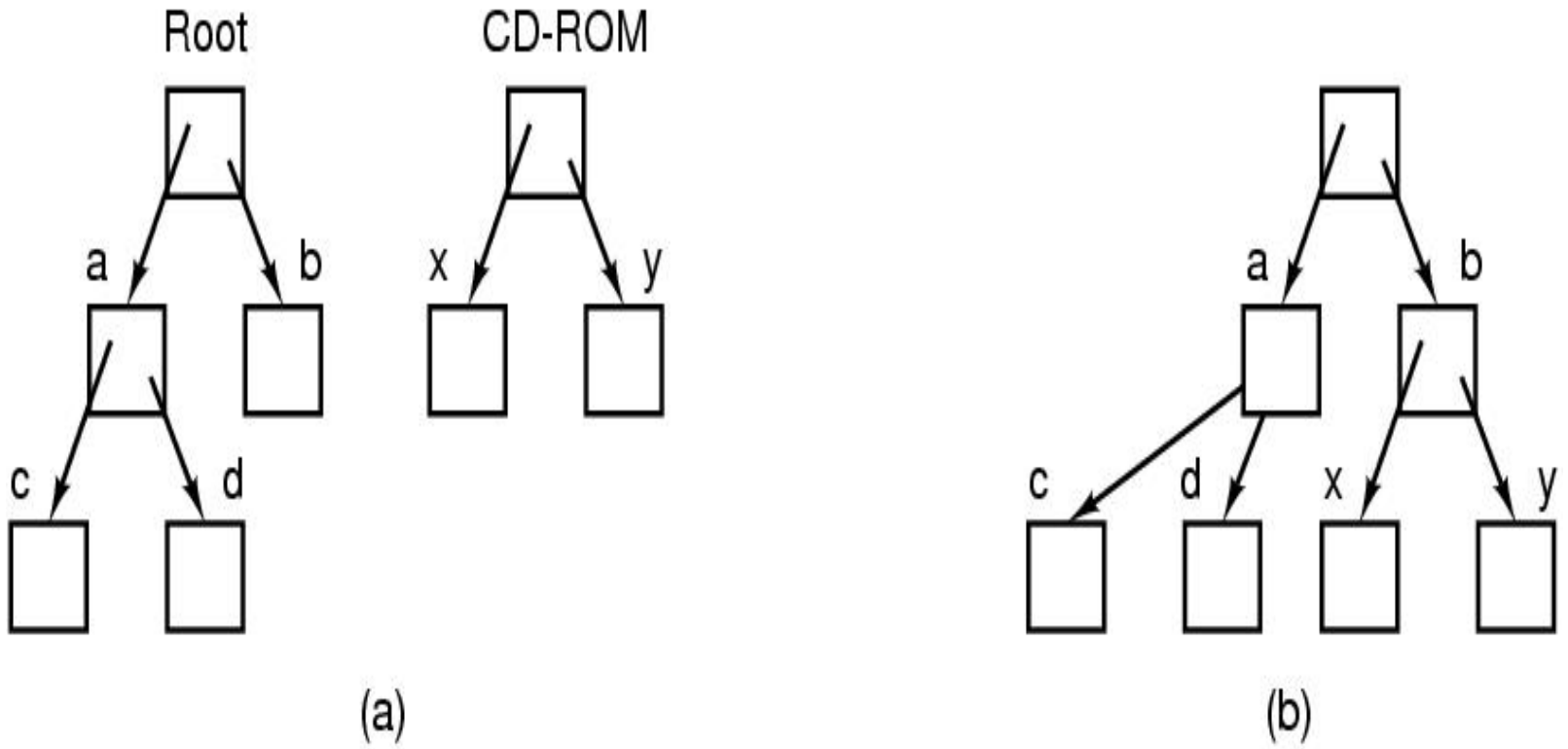


(a)



(b)

Example – File System Mounting



(a) Before mounting (b) Mounted system

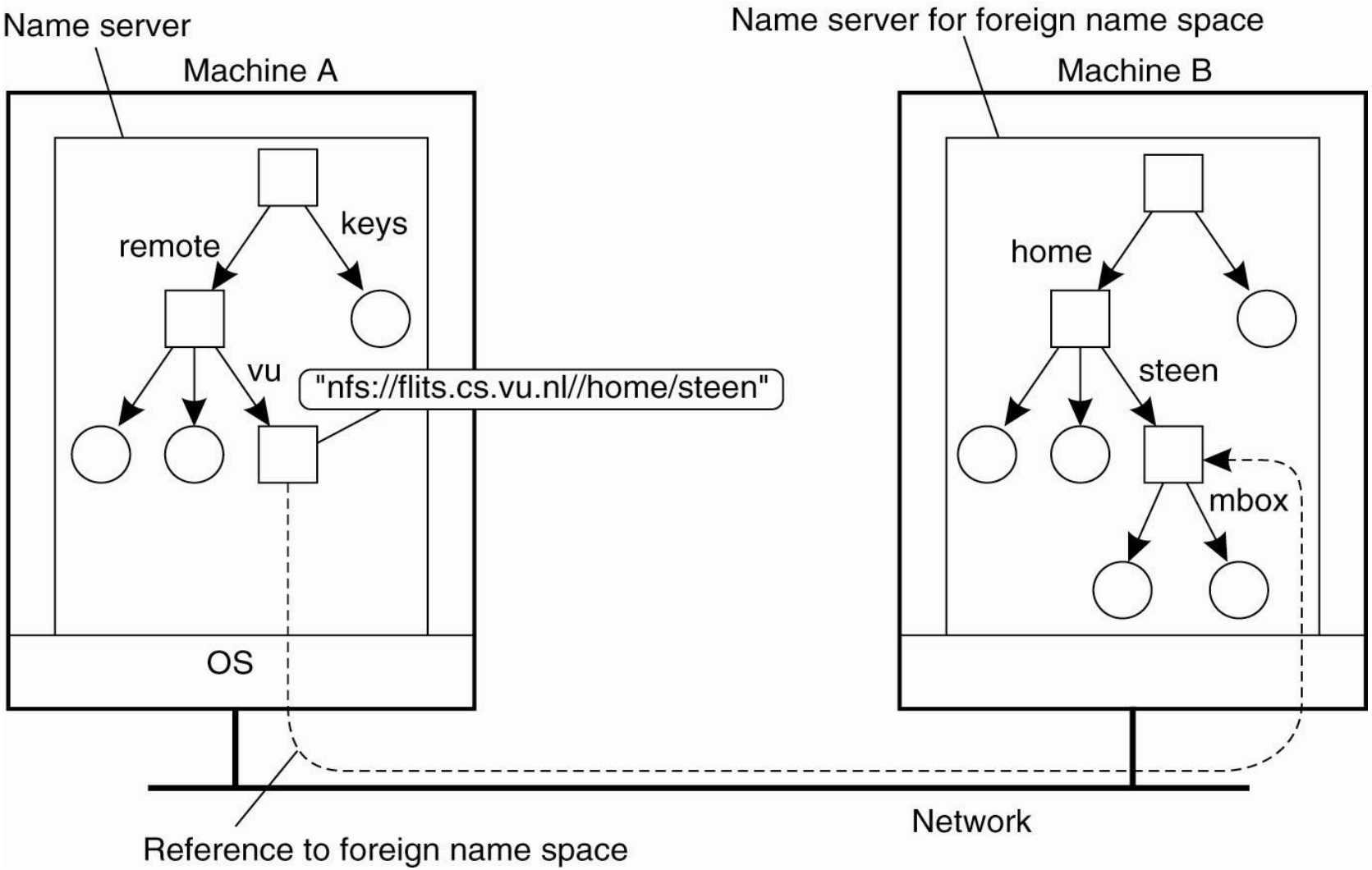
File Sharing

- Sharing of files on multi-user systems is desirable.
- Sharing may be done through a *protection* scheme.
- On distributed systems, files may be shared across a network.
- Network File System (NFS) is a common distributed file-sharing method.
- If multi-user system:
  - **User IDs** identify users, allowing permissions and protections to be per-user.
  - **Group IDs** allow users to be in groups, permitting group access rights.
  - Owner/Group of a file/directory.

File Sharing – Remote File Systems

- Uses networking to allow file system access between systems:
  - Manually via programs like FTP.
  - Automatically, seamlessly using **distributed file systems.**
  - Semi automatically via the **Web.**
- Client-server model allows clients to mount remote file systems from servers:
  - Server can serve multiple clients.
  - Client & user-on-client identification is insecure/complicated.
  - **NFS** is standard UNIX client-server file sharing protocol.
  - **CIFS** is standard Windows protocol.
  - Standard operating system file calls are translated into remote calls.
  - Distributed Information Systems (distributed naming services) such as LDAP, DNS, NIS, Active Directory implement unified access to information needed for remote computing.

Mounting Remote Name Spaces



File Sharing – Failure Modes

- All file systems have failure modes:
  - For example corruption of directory structures or other non-user data, called metadata.
- Remote file systems add new failure modes, due to network failure, server failure.
- Recovery from failure can involve state information about status of each remote request.
- Stateless protocols such as NFS v3 include all information in each request, allowing easy recovery but less security.

Protection

- File owner/creator should be able to control:
  - what can be done
  - and by whom
- Types of access:
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

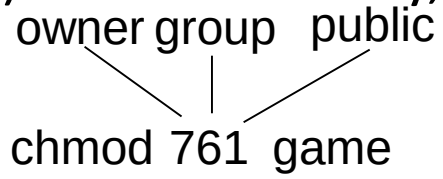
Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users:

			RWX
a) **owner access**	7	⇒	1 1 1
			RWX
b) **group access**	6	⇒	1 1 0
			RWX
c) **public access**	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.
- Attach a group to a file

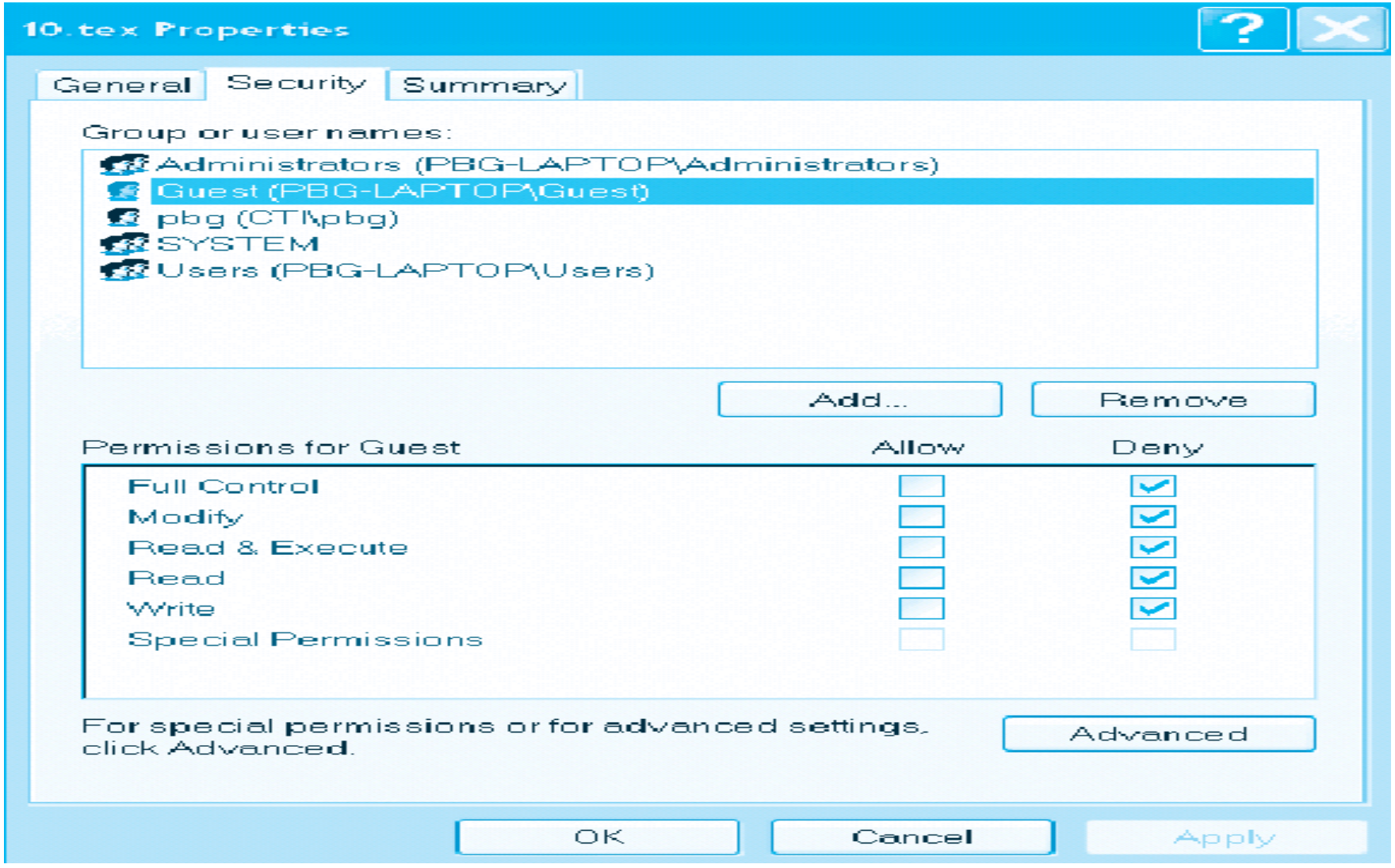
chgrp G game



A Sample UNIX Directory Listing

-rw-rw-r--	1 pbg	staff	31200	Sep 3 08:30	intro.ps
drwx------	5 pbg	staff	512	Jul 8 09.33	private/
drwxrwxr-x	2 pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2 pbg	student	512	Aug 3 14:13	student-proj/
-rw-r--r--	1 pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1 pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4 pbg	faculty	512	Jul 31 10:31	lib/
drwx------	3 pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

Windows XP
Access-control List Management



ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

File System Implementation

SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

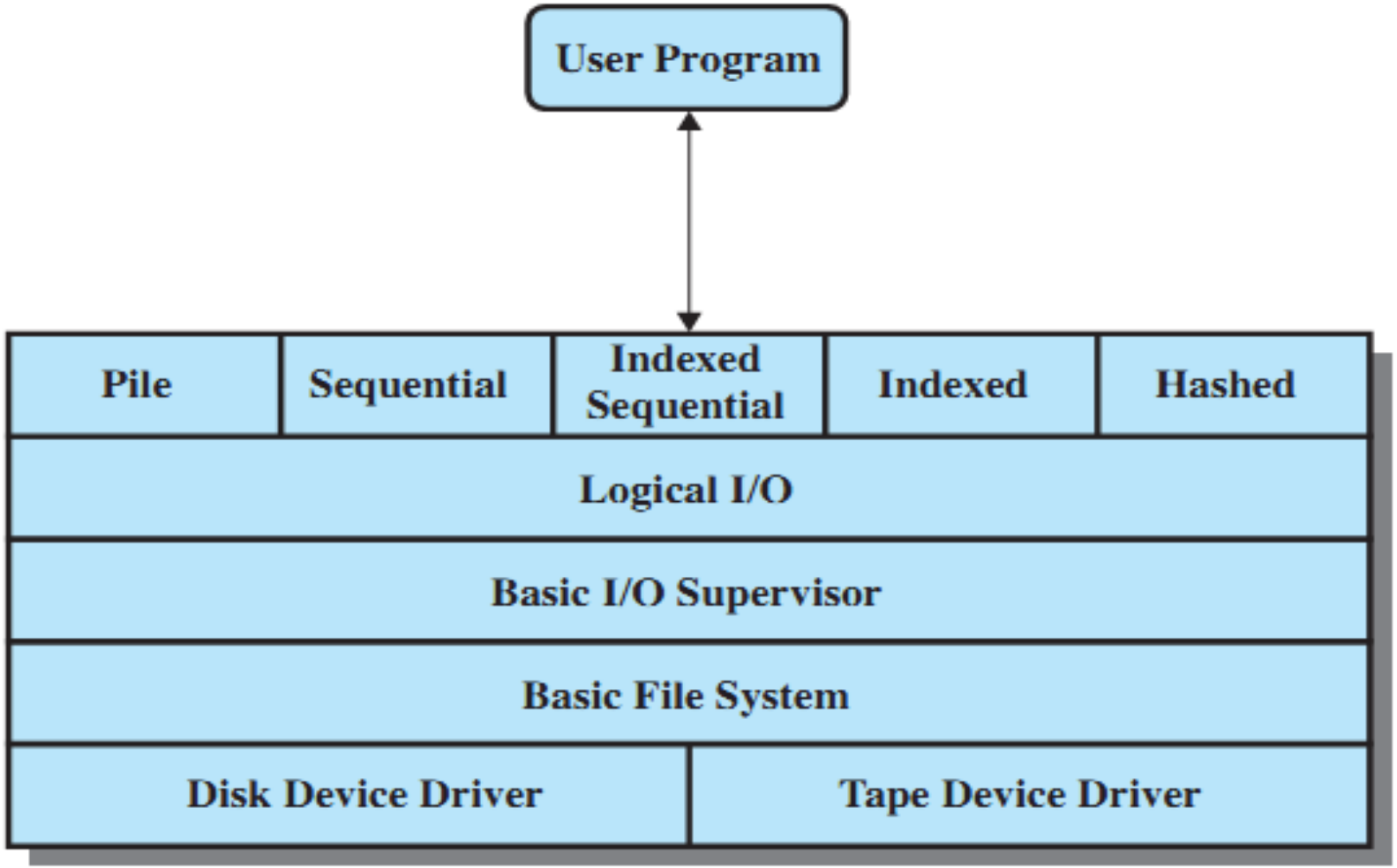
File System Implementation

- File-System Structure
- File-System Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Recovery

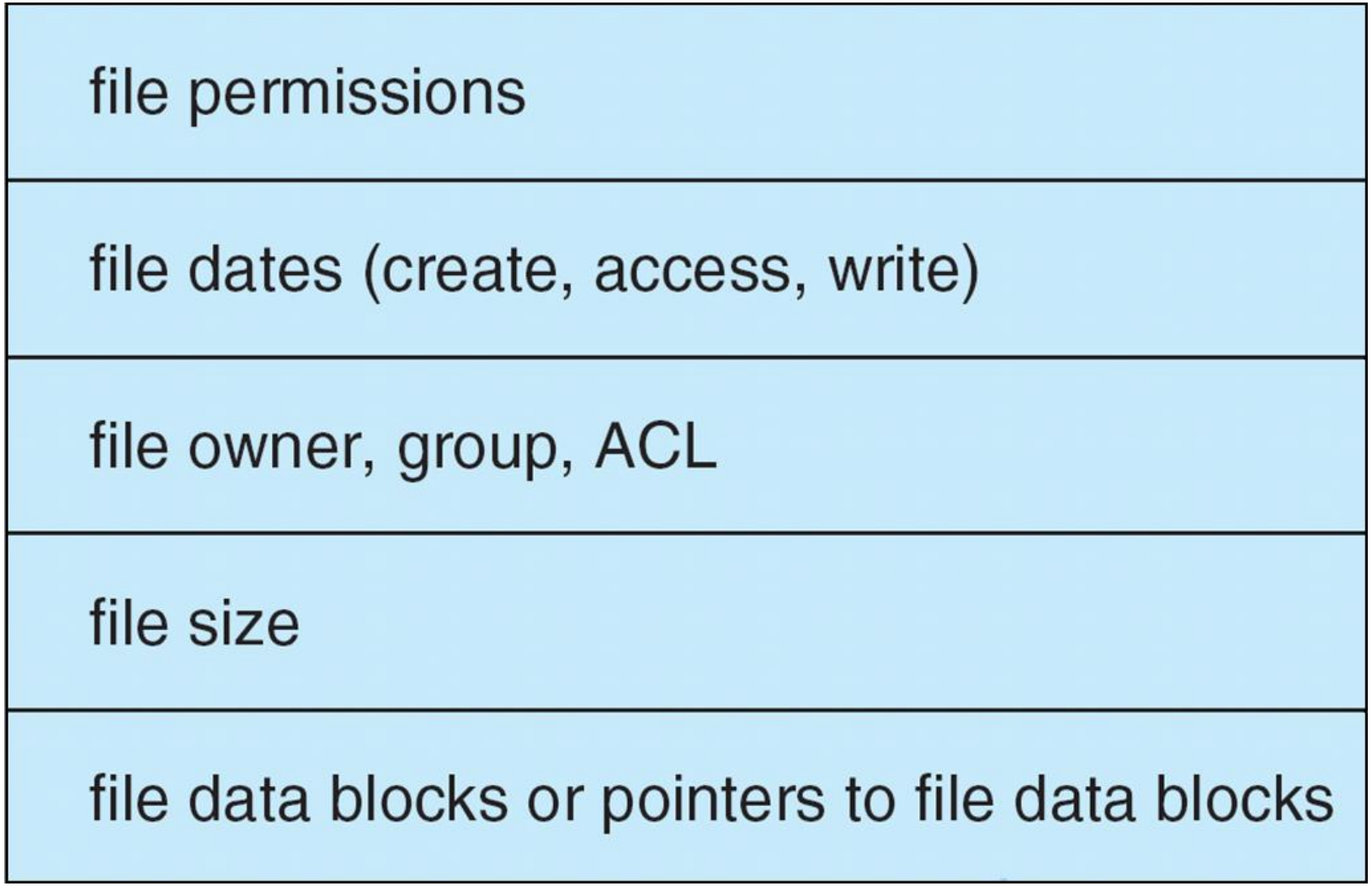
File-System Structure

- File structure:
  - Logical storage unit
  - Collection of related information
- File system resides on secondary storage (disks):
  - Provided user interface to storage, mapping logical to physical.
  - Provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily.
- Disk provides in-place rewrite and random access:
  - I/O transfers performed in blocks of sectors (usually 512 bytes).
- File control block – storage structure consisting of information about a file.
- Device driver controls the physical device.
- File system organized into layers.

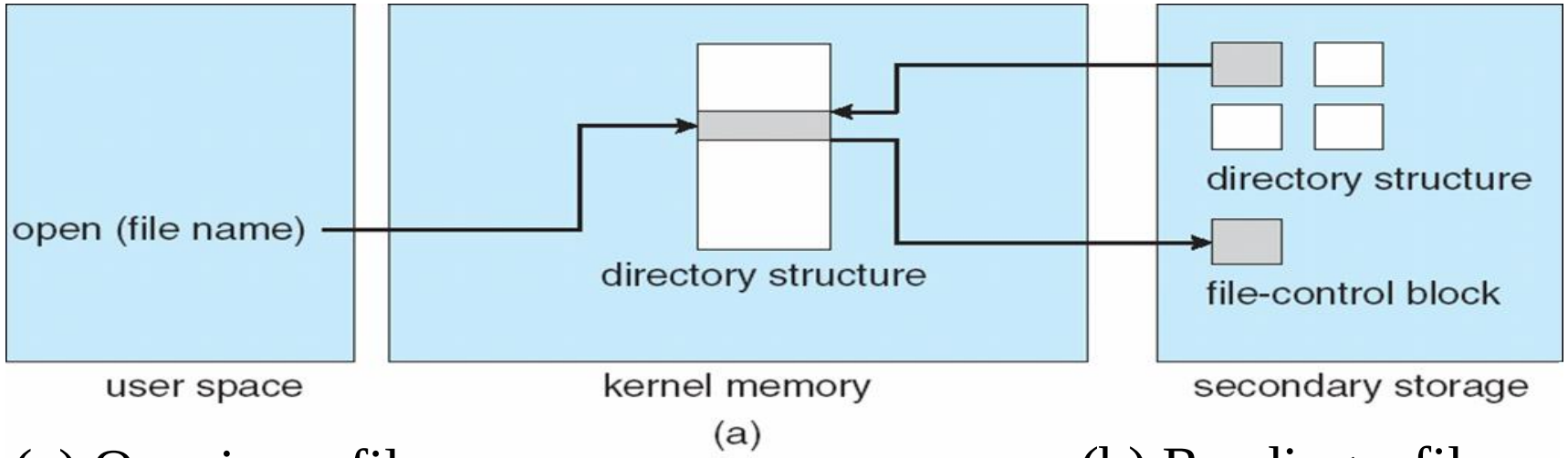
File System Software Architecture



A Typical File Control Block (FCB)

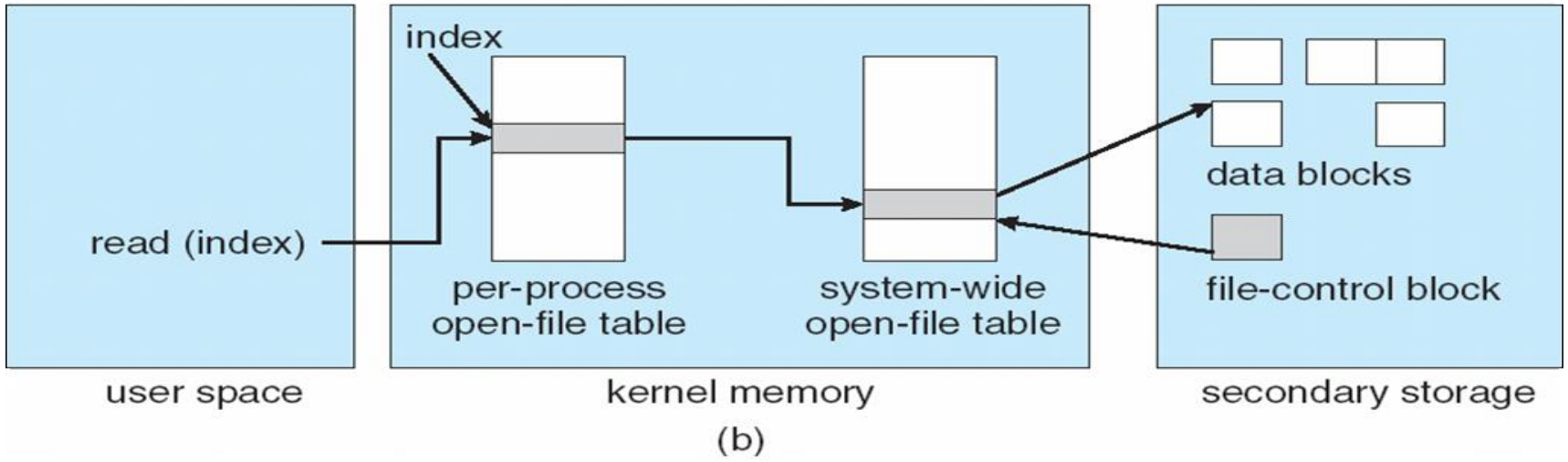


In-Memory File System Structures



(a) Opening a file

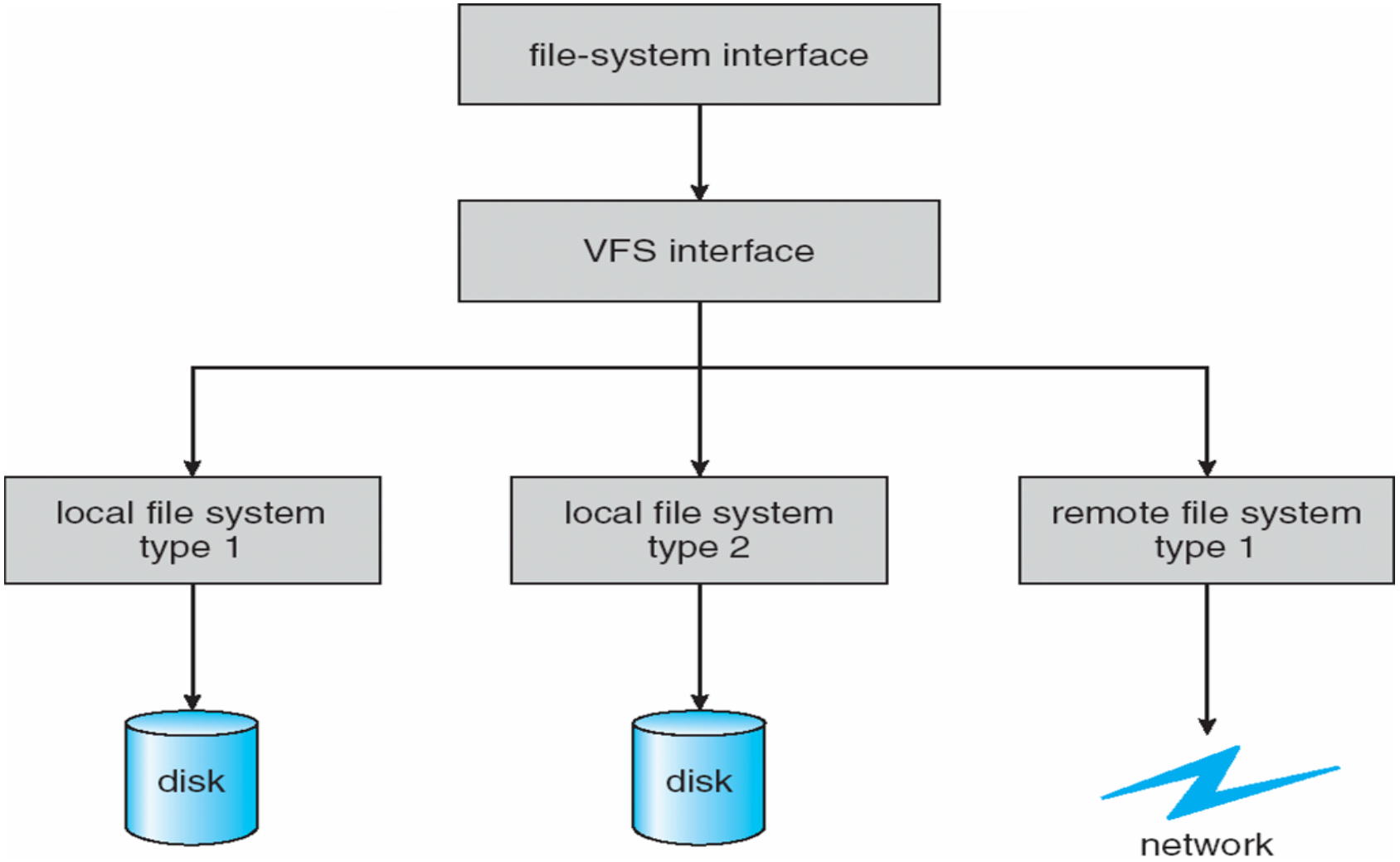
(b) Reading a file



Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

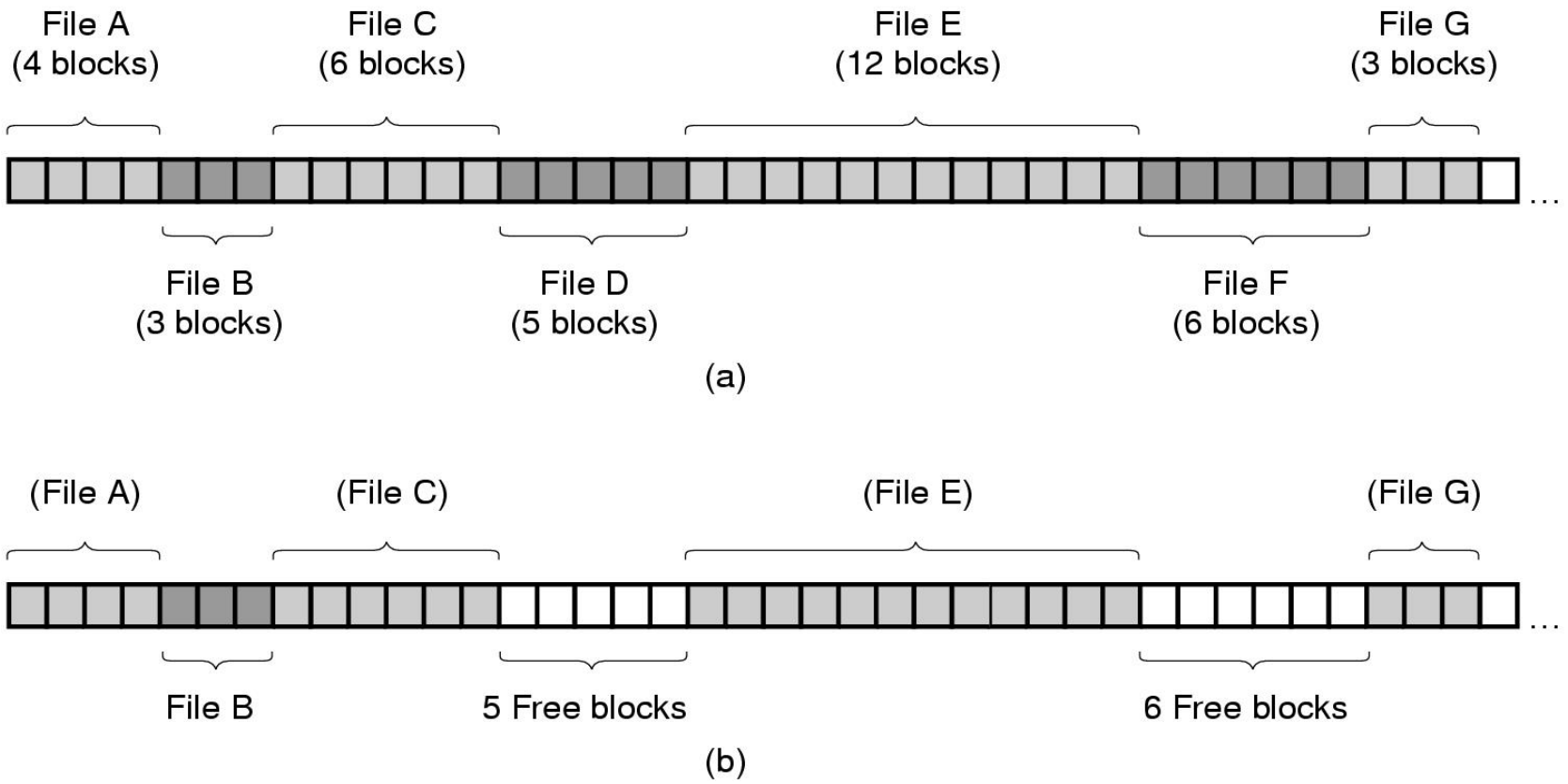
Schematic View of Virtual File System



Allocation Methods

- An allocation method refers to how disk blocks are allocated for files:
  1. Contiguous allocation
  2. Chained/Linked allocation
  3. Indexed allocation
  4. Combined schemes

Contiguous Allocation Example

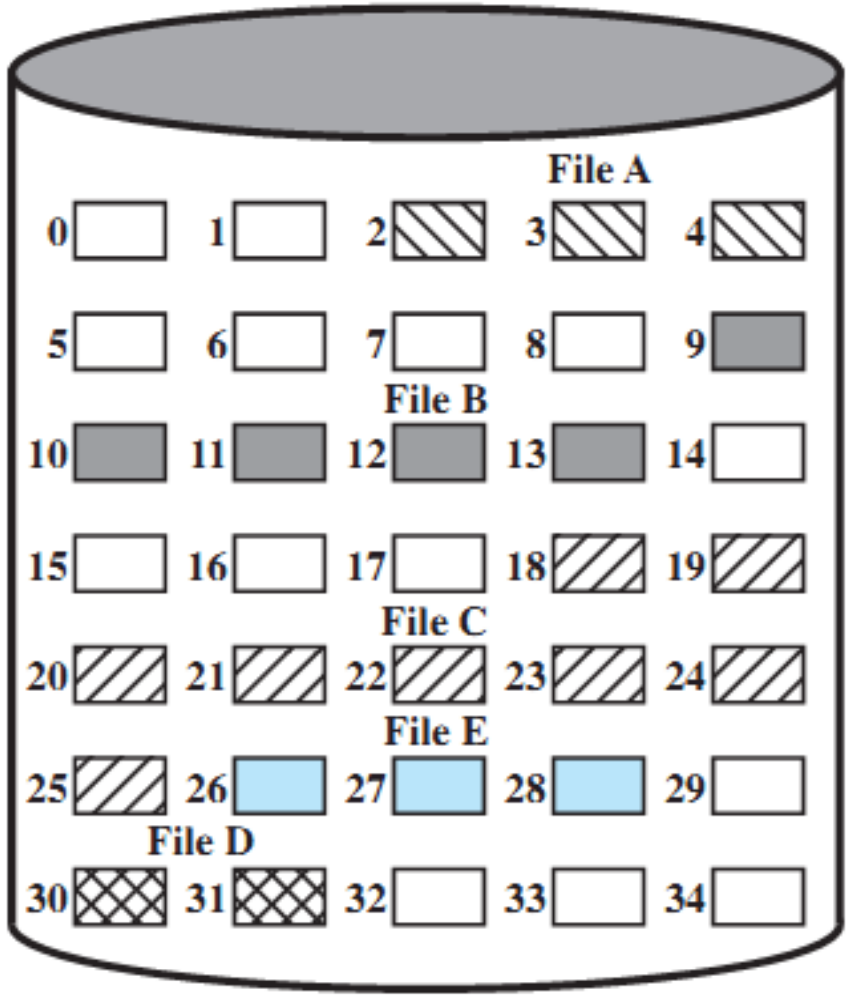


(a) Contiguous allocation of disk space for 7 files.
(b) The state of the disk after files D and F have been removed.

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk.
- Simple: only starting location (block #) and length (number of blocks) required.
- Enables random access.
- Wasteful of space (dynamic storage-allocation problem).
- Files cannot grow.

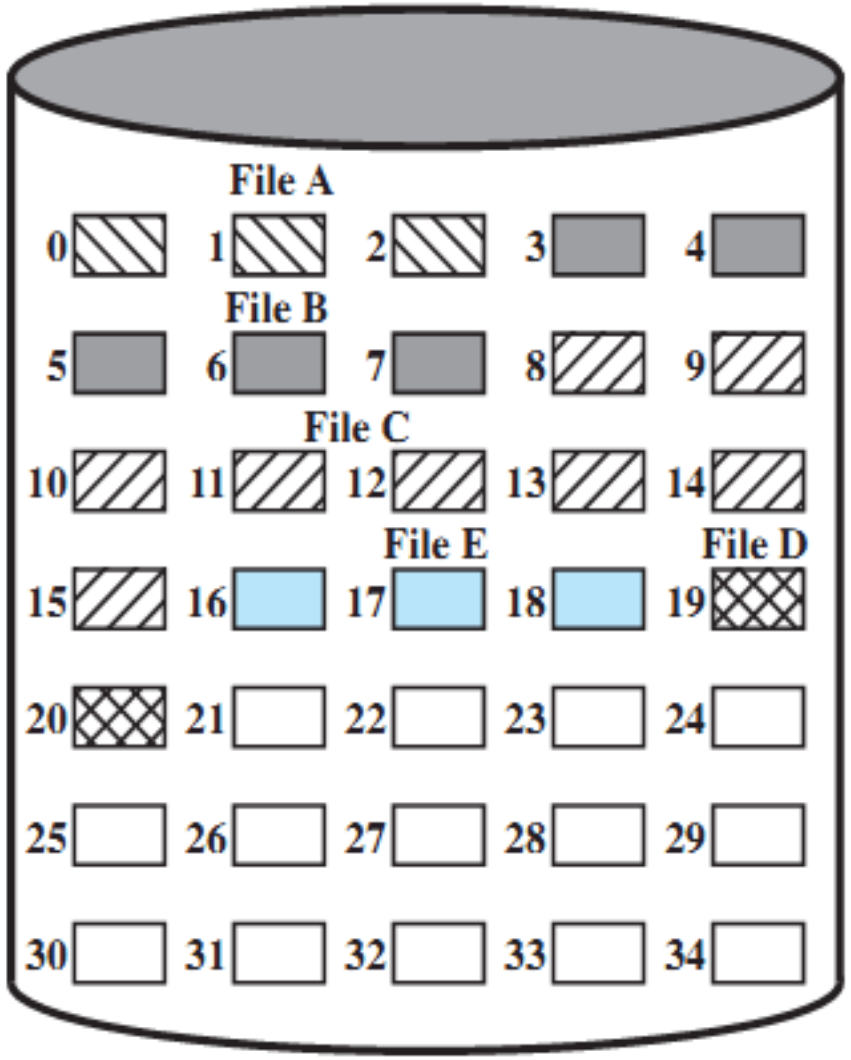
Contiguous File Allocation Example



File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Contiguous file allocation (after compaction)



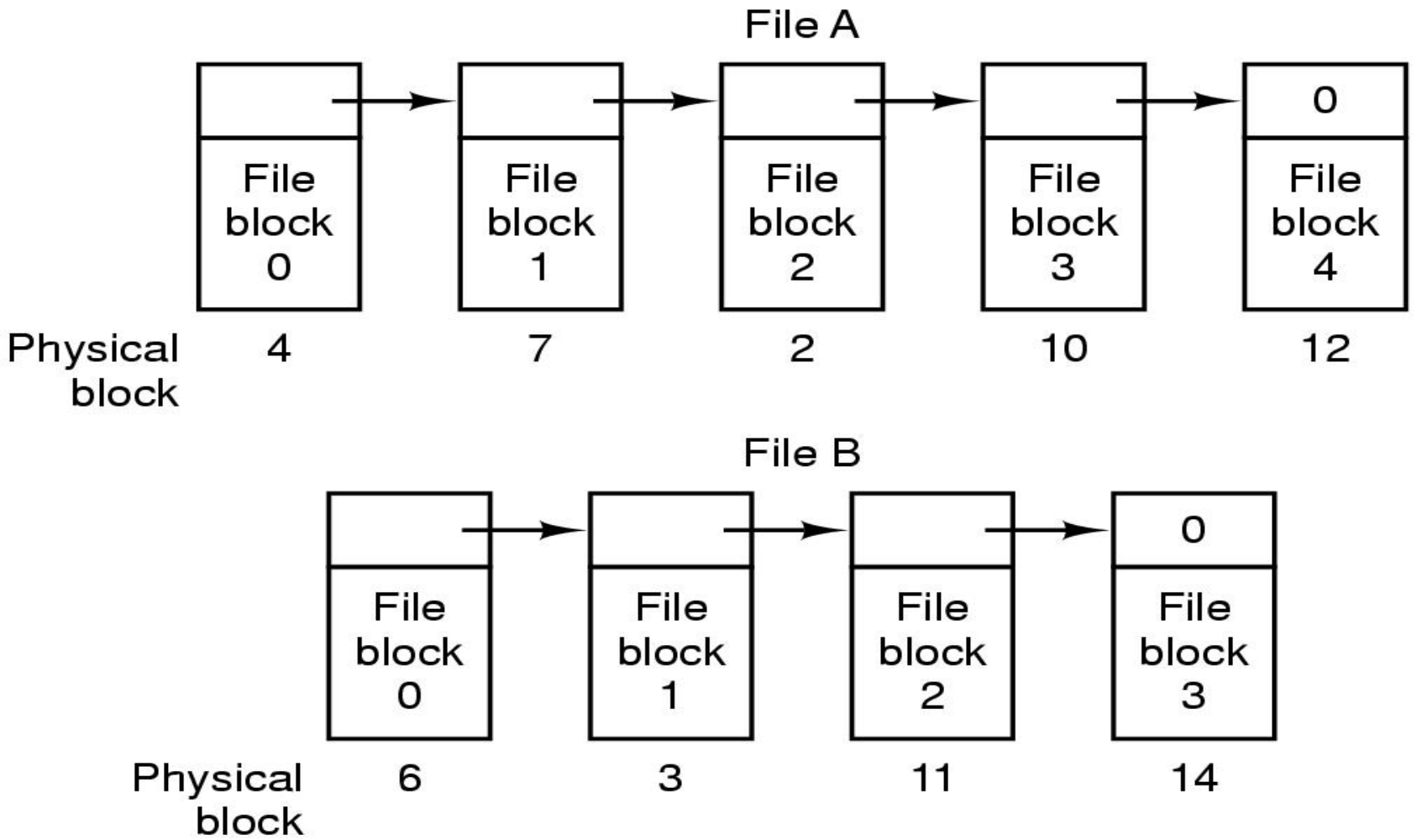
File Allocation Table

File Name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Extent-Based Systems

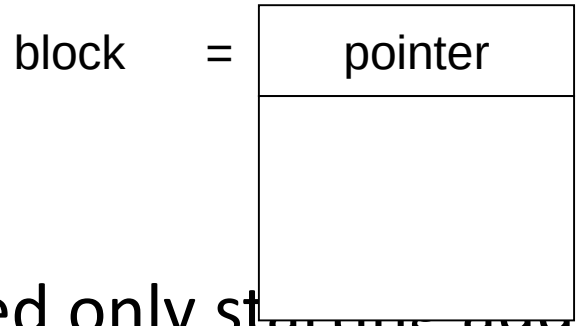
- Many newer file systems (i.e., Veritas File System) use a modified contiguous allocation scheme.
- Extent-based file systems allocate disk blocks in extents.
- An extent is a contiguous block of disks:
  - Extents are allocated for file allocation.
  - A file consists of one or more extents.

Chained/Linked Allocation Example



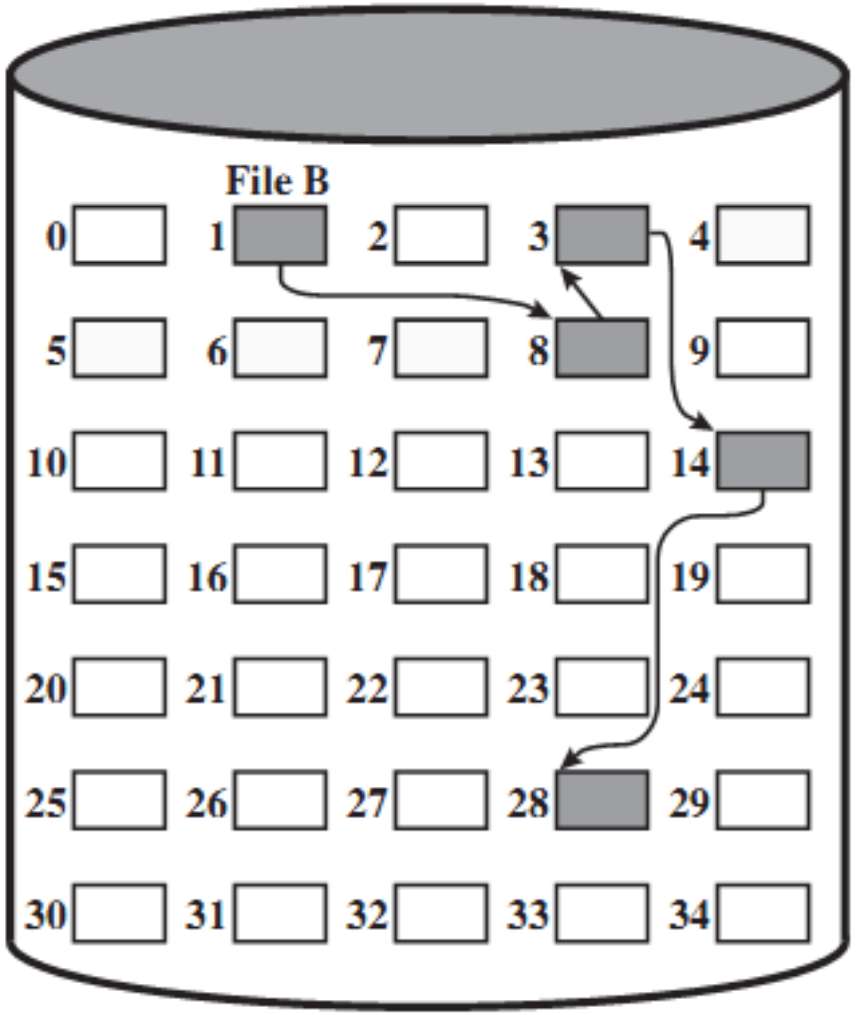
Chained Allocation

- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.



- Simple: need only starting address.
- Free-space management: no waste of space.
- No random access.

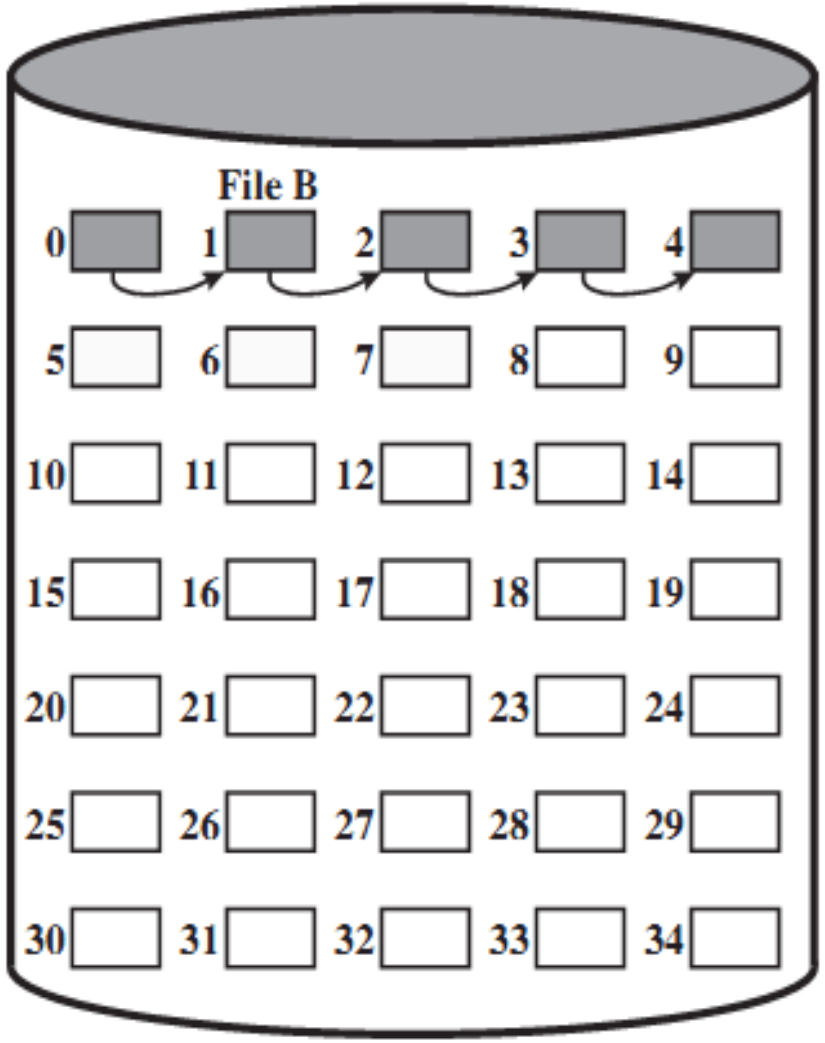
Chained File Allocation Example



File Allocation Table

File Name	Start Block	Length
• • •	• • •	• • •
File B	1	5
• • •	• • •	• • •

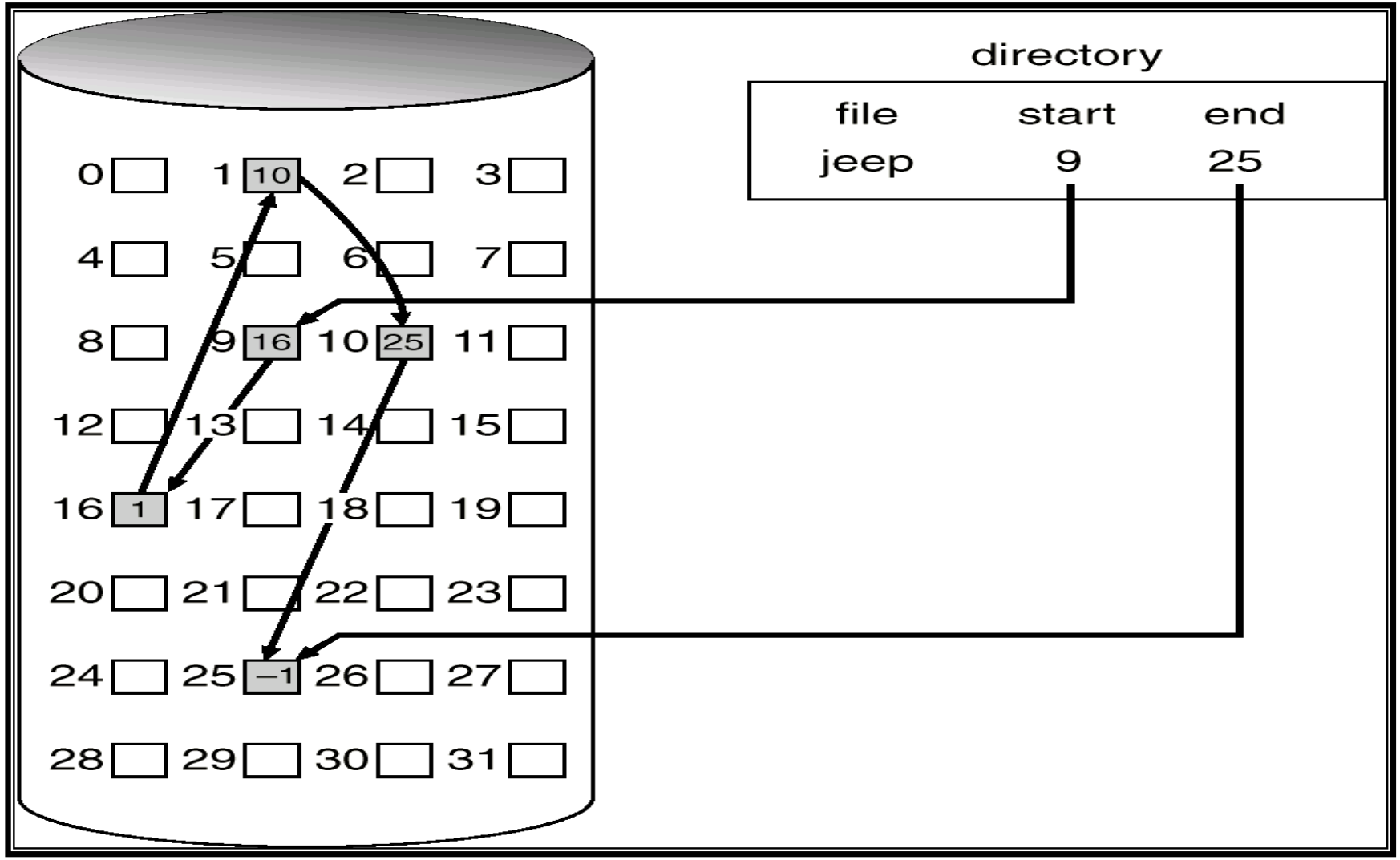
Chained File Allocation (after consolidation)



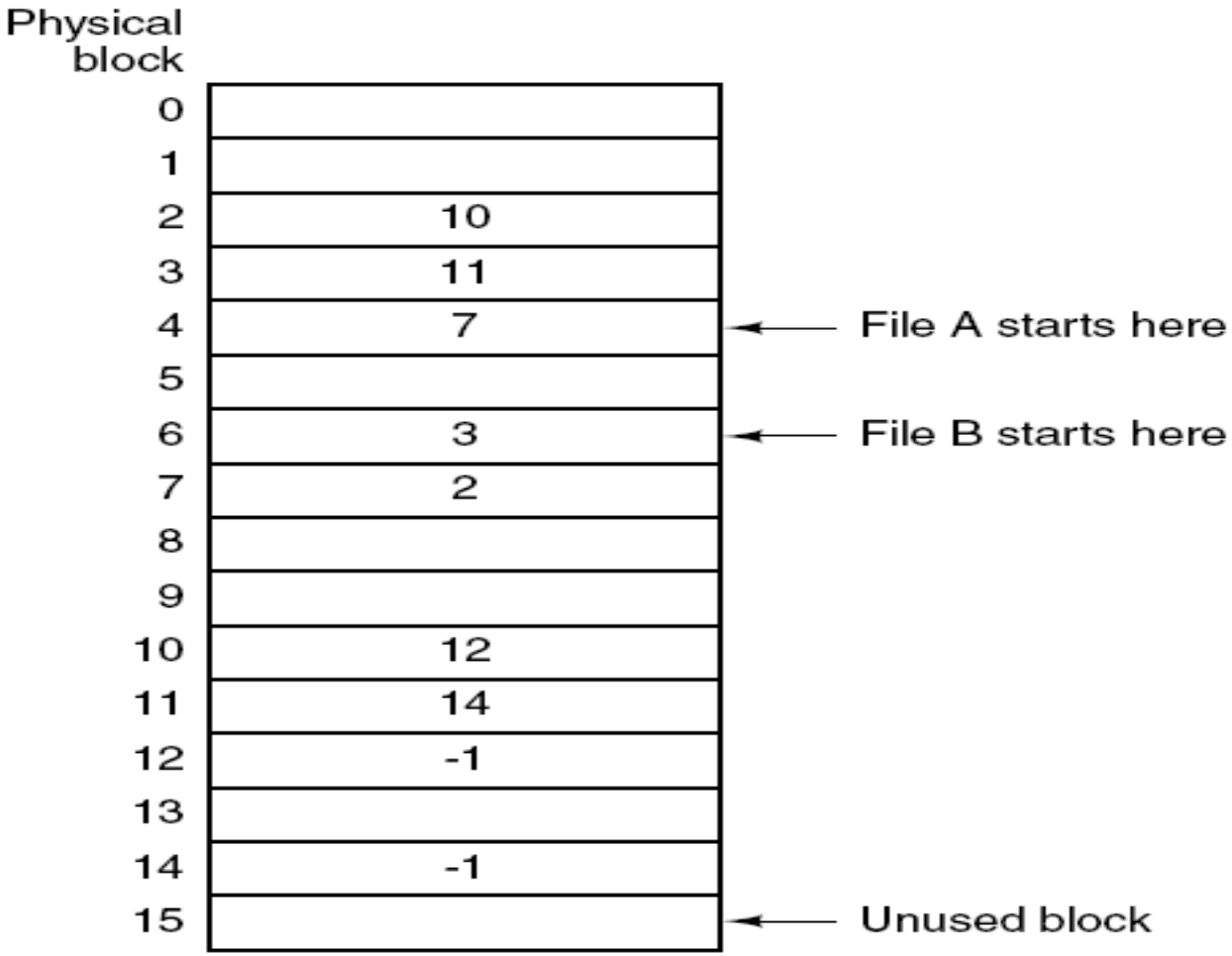
File Allocation Table

File Name	Start Block	Length
• • •	• • •	• • •
File B	0	5
• • •	• • •	• • •

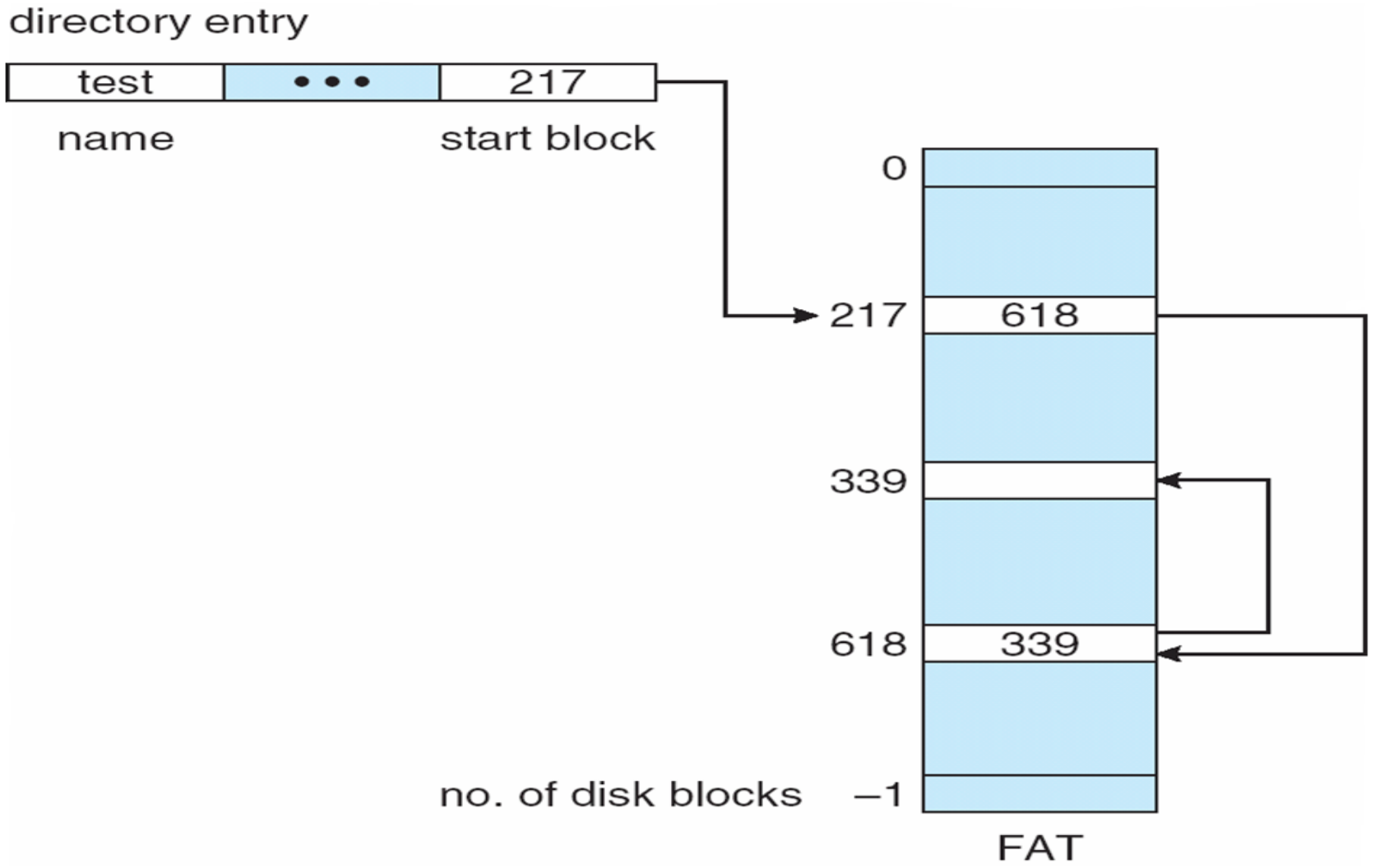
Another Chained Allocation Example



Linked List Allocation Using a Table in Memory

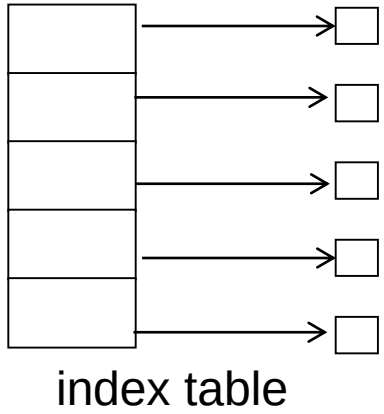


File-Allocation Table (FAT) Example

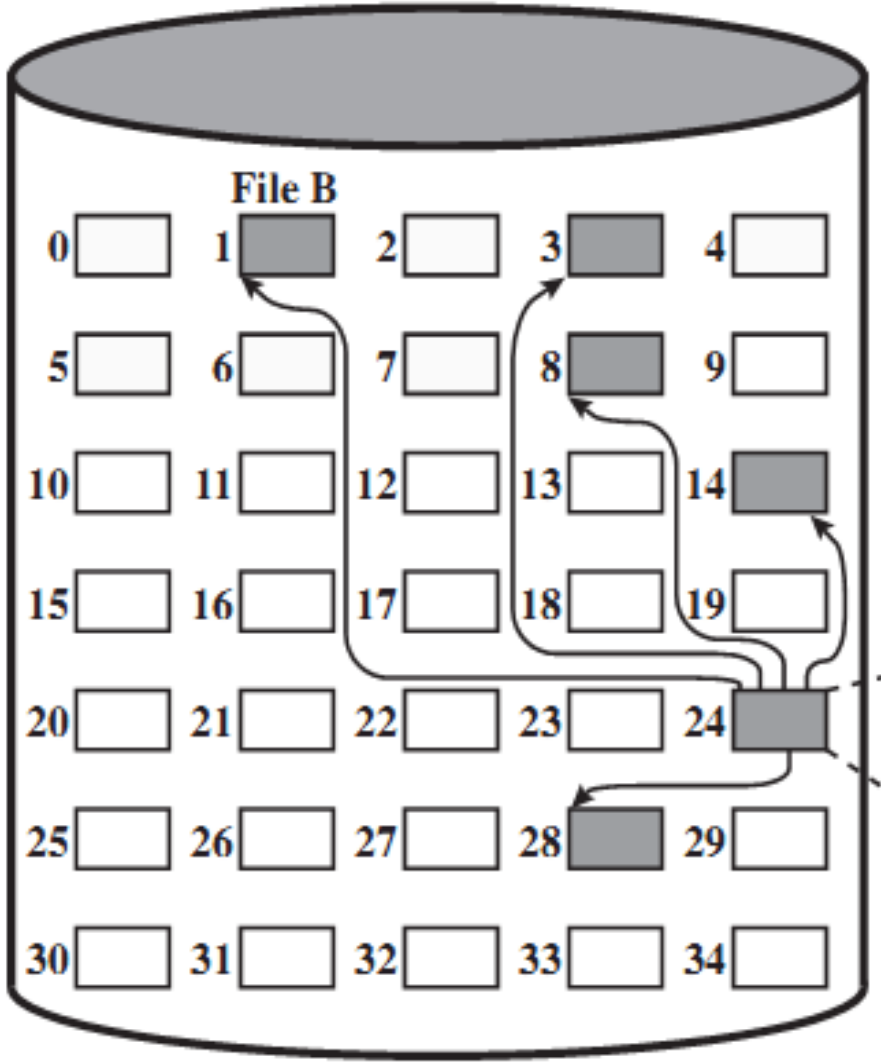


Indexed Allocation

- Brings all pointers together into the *index block.*
- It’s a logical view.
- Need index table.
- Provides random access.
- Dynamic access without external fragmentation, but have overhead of index block.



Indexed File Allocation Example

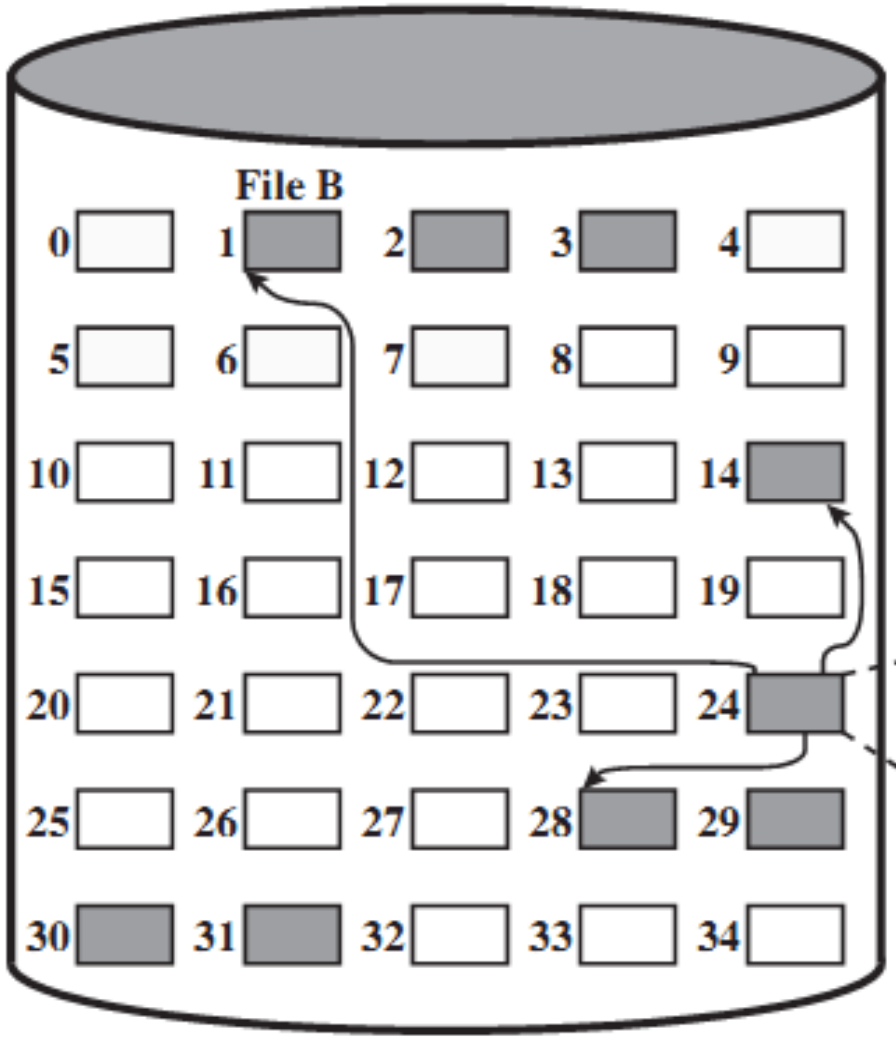


File Allocation Table

File Name	Index Block
• • •	• • •
File B	24
• • •	• • •

1
8
3
14
28

Indexed File Allocation (variable-size)

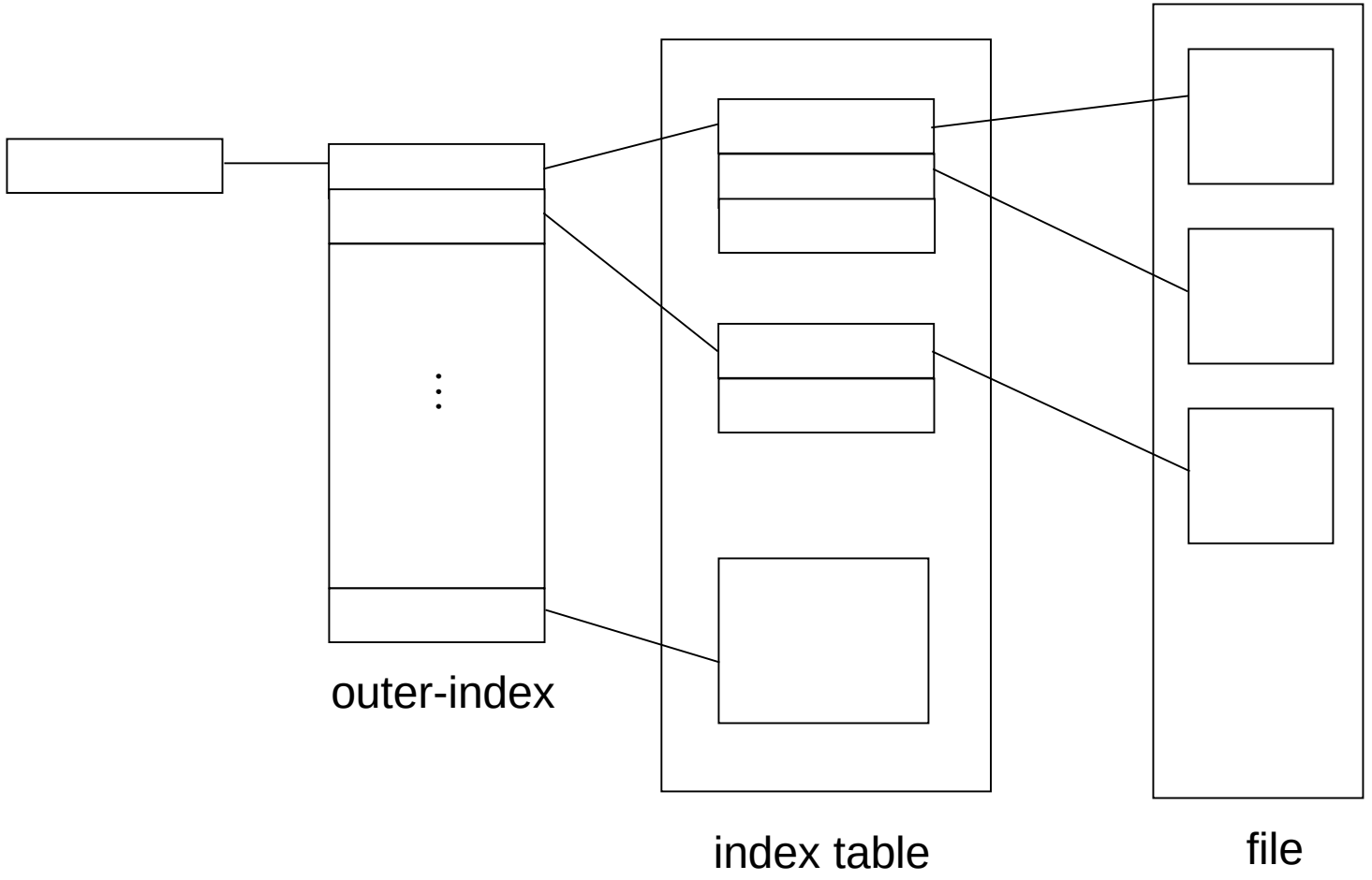


File Allocation Table

File Name	Index Block
• • •	• • •
File B	24
• • •	• • •

Start Block	Length
1	3
28	4
14	1

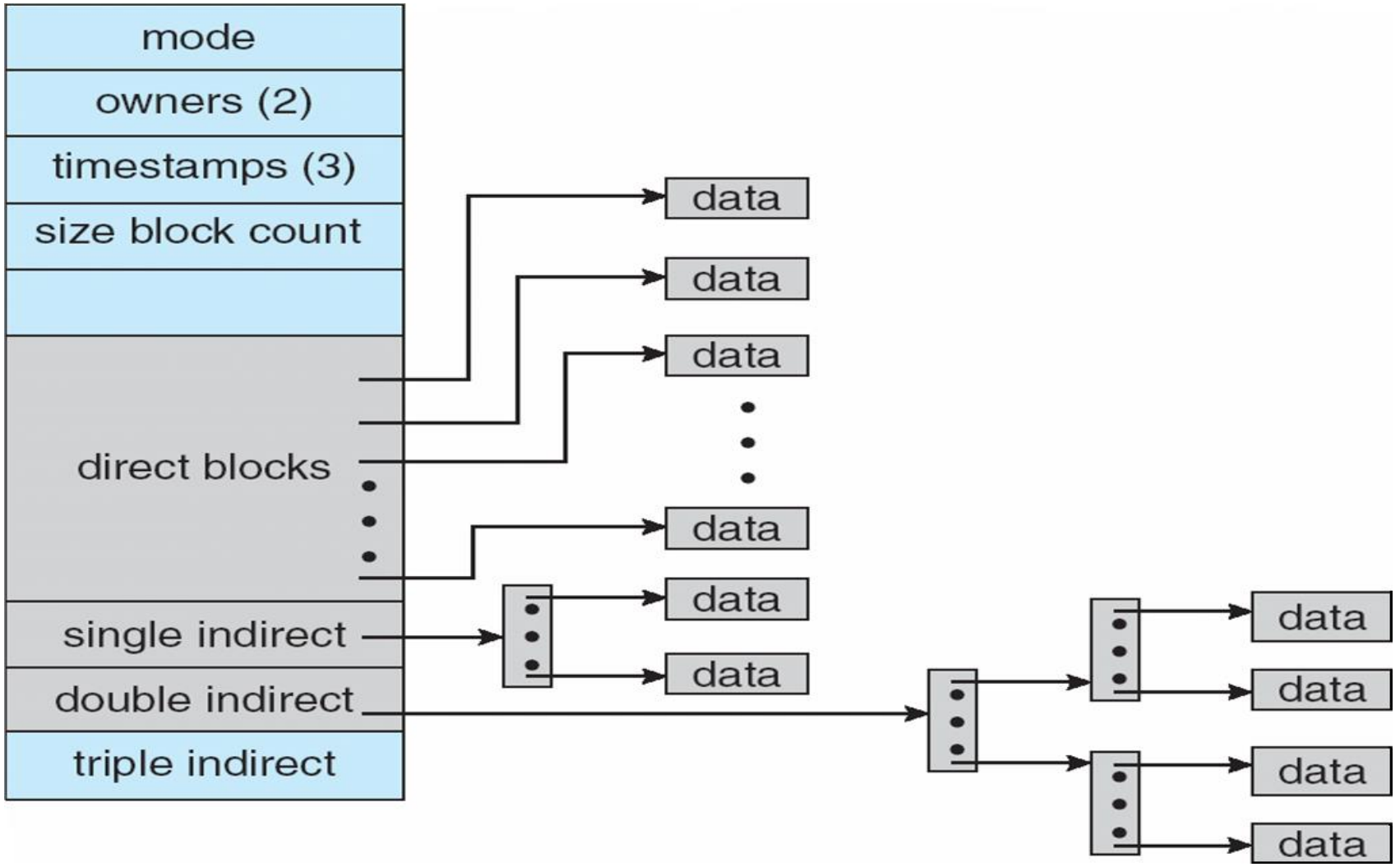
Indexed Allocation Mapping Example



Comparison of file allocation methods

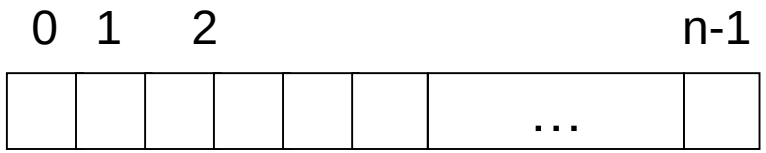
	Contiguous	Chained	Indexed	
Preallocation?	Necessary	Possible	Possible	
Fixed or variable size portions?	Variable	Fixed blocks	Fixed blocks	Variable
Portion size	Large	Small	Small	Medium
Allocation frequency	Once	Low to high	High	Low
Time to allocate	Medium	Long	Short	Medium
File allocation table size	One entry	One entry	Large	Medium

Combined Scheme: UNIX UFS (4K bytes per block)



Free-Space Management (1)

- Bit vector (*n* blocks)



$$bit[i] = \begin{cases} 0 \Rightarrow block[i] \text{ free} \\ 1 \Rightarrow block[i] \text{ occupied} \end{cases}$$

Block number calculation

(number of bits per word) *
(number of 0-value words) +
offset of first 1 bit

Free-Space Management (2)

- Bit map requires extra space. For example:

  block size = 2^{12} bytes

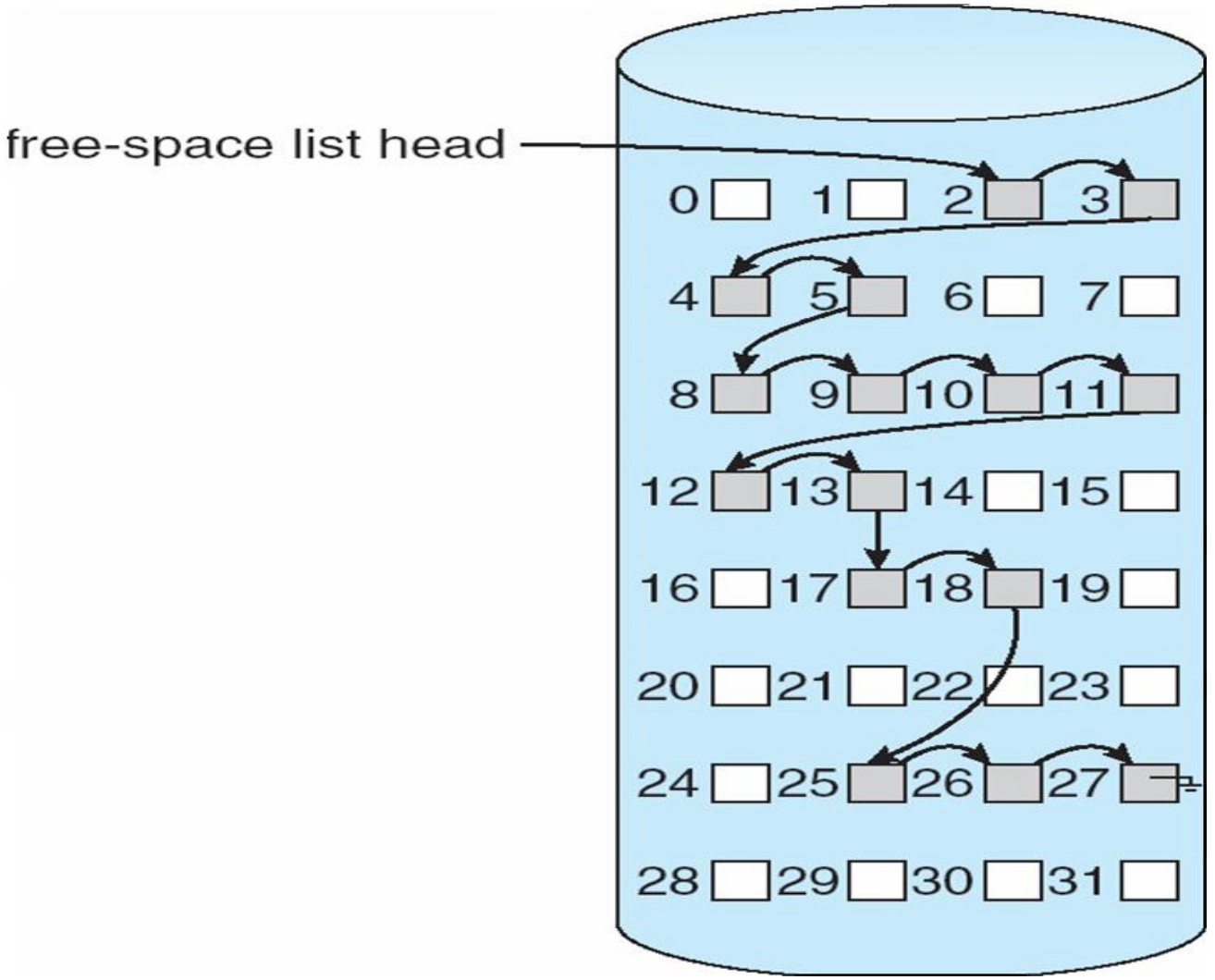
  disk size = 2^{30} bytes (1 gigabyte)

  $n = 2^{30}/2^{12} = 2^{18}$ bits (or 32K bytes)
- Easy to get contiguous files.
- Linked list (free list):
  - Cannot get contiguous space easily.
  - No waste of space.
- Grouping
- Counting

Free-Space Management (3)

- Need to protect:
  - Pointer to free list.
  - Bit map:
    - Must be kept on disk.
    - Copy in memory and disk may differ.
    - Cannot allow for block[*i*] to have a situation where bit[*i*] = 1 in memory and bit[*i*] = 0 on disk.
  - Solution:
    - Set bit[*i*] = 1 in disk.
    - Allocate block[*i*].
    - Set bit[*i*] = 1 in memory.

Linked Free Space List on Disk



Directory Implementation

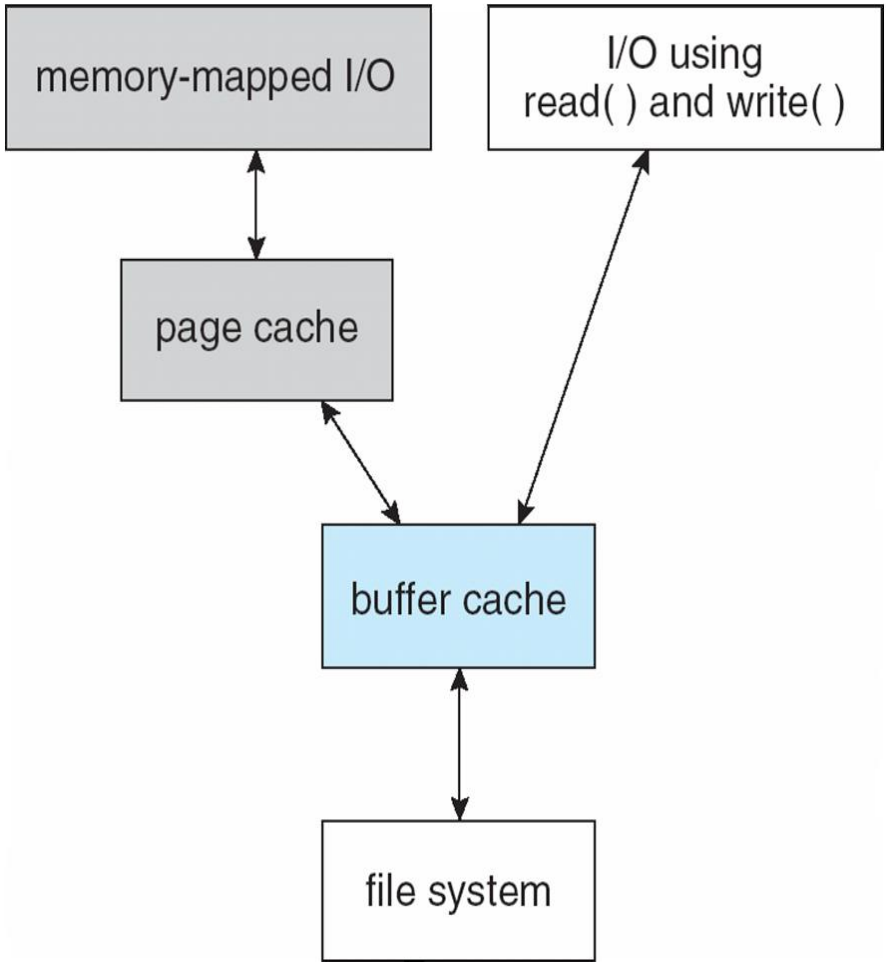
- Linear list of file names with pointer to the data blocks:
  - simple to program
  - time-consuming to execute
- Hash Table – linear list with hash data structure:
  - decreases directory search time
  - *collisions* – situations where two file names hash to the same location
  - fixed size

Efficiency and Performance

- Efficiency dependent on:
  - Disk allocation and directory algorithms.
  - Types of data kept in file's directory entry.
- Performance:
  - Disk cache: separate section of main memory for frequently used blocks.
  - Free-behind and read-ahead: techniques to optimize sequential access.
  - Improve PC performance by dedicating section of memory as virtual disk or RAM disk.

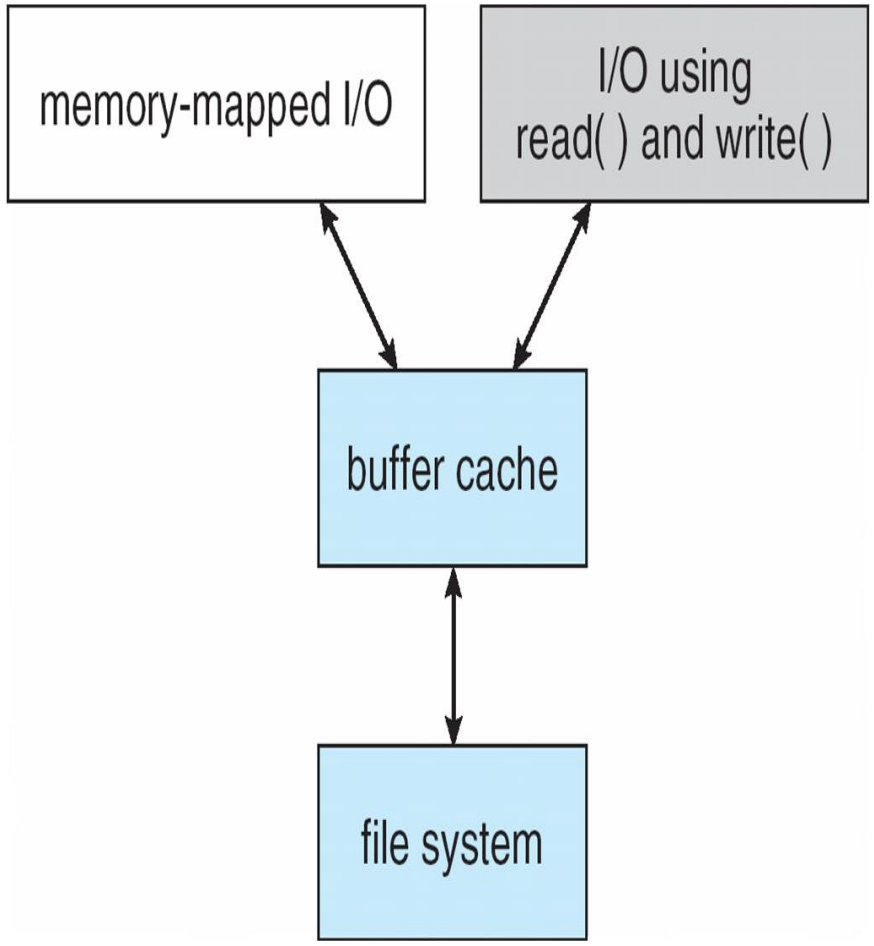
Page Cache

- A **page cache** caches pages rather than disk blocks using virtual memory techniques.
- Memory-mapped I/O uses a page cache.
- Routine I/O through the file system uses the buffer (disk) cache.



Unified Buffer Cache

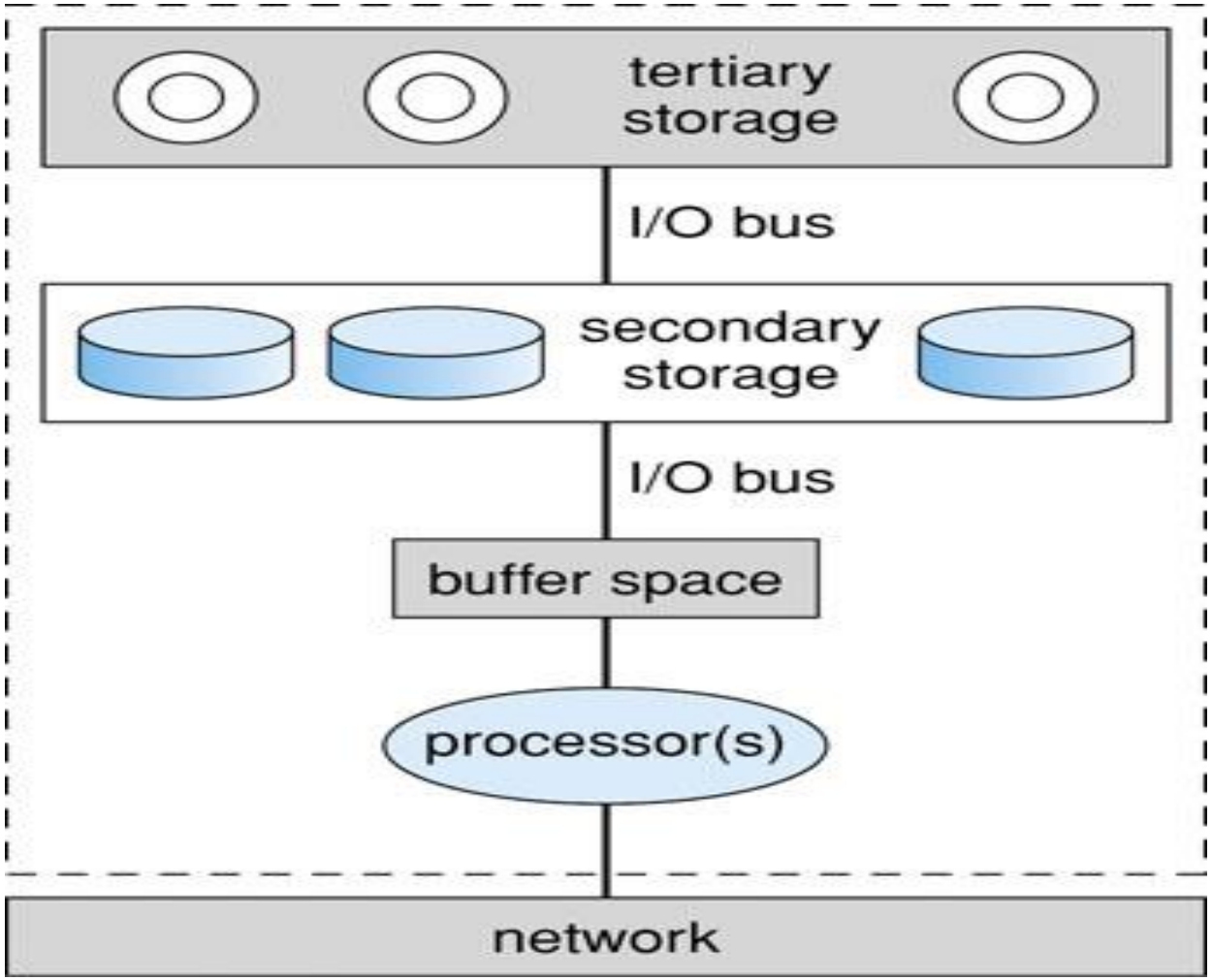
- A unified buffer cache uses the same page cache to cache both memory-mapped pages and ordinary file system I/O.



Recovery

- Consistency checking – compares data in directory structure with data blocks on disk,
  and tries to fix inconsistencies.
- Use system programs to back up data from disk to another storage device (floppy disk, magnetic tape).
- Recover lost file or disk by restoring data from backup.

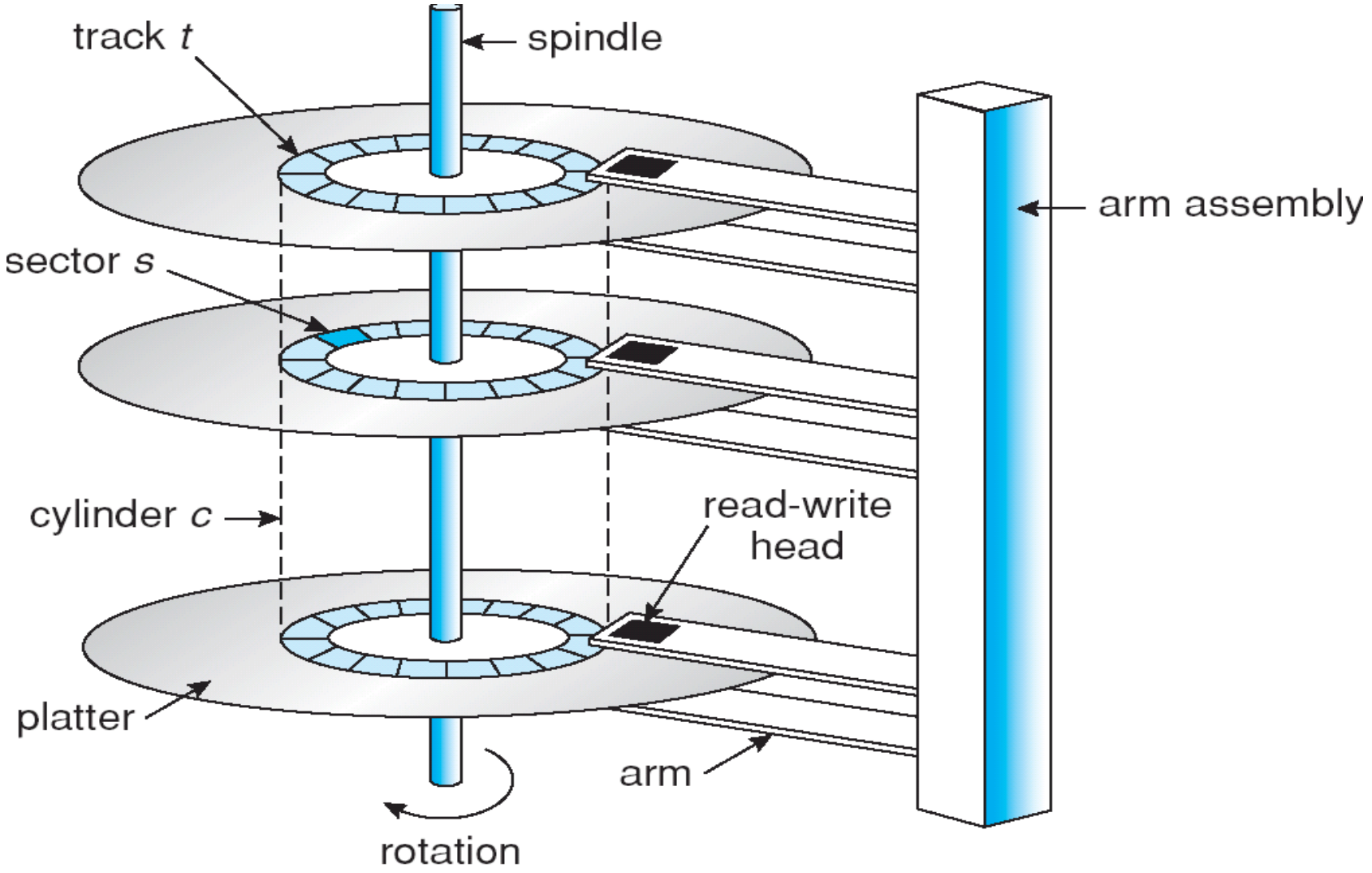
Storage Backup



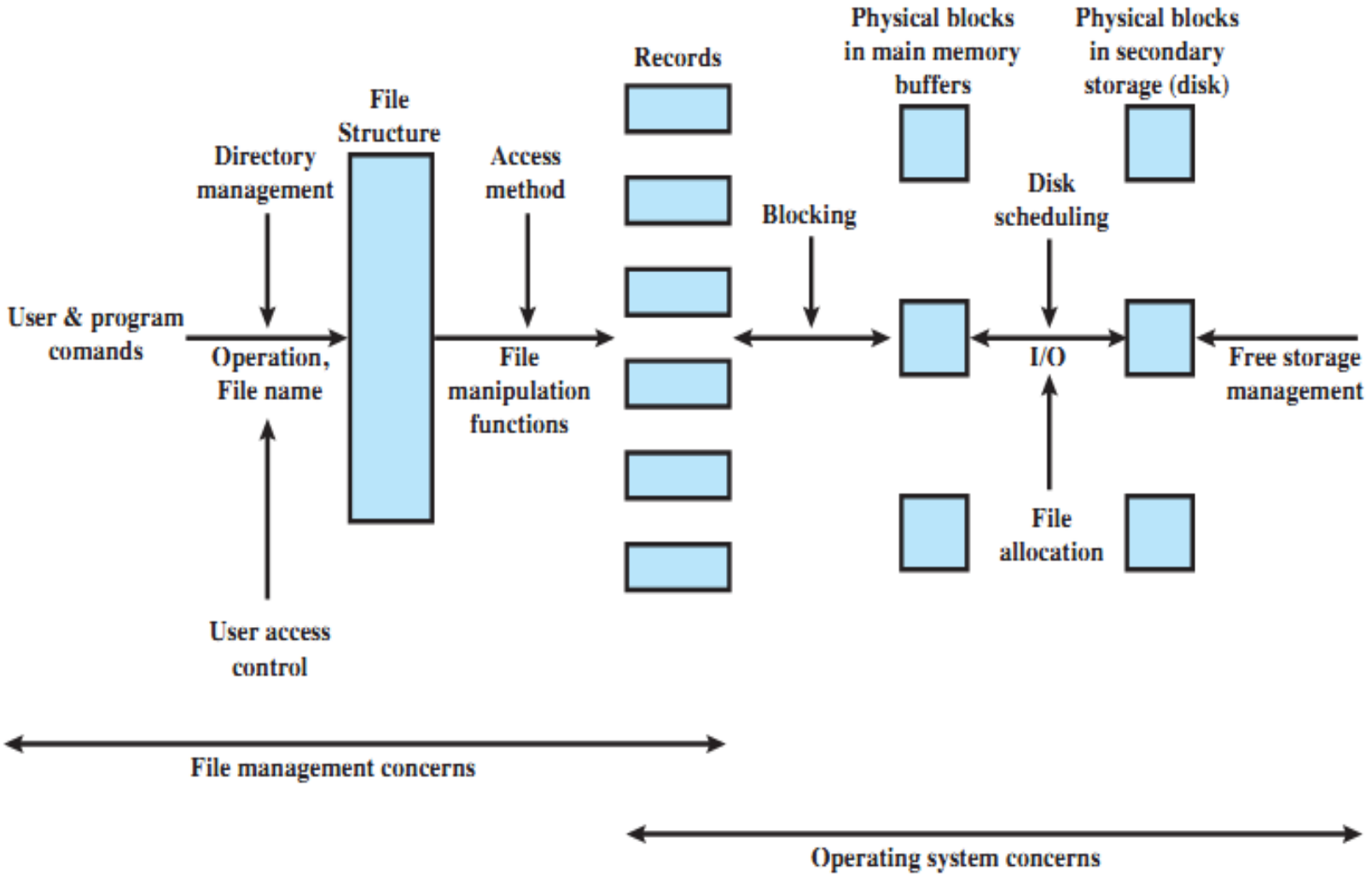
ie
SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY

Disk Scheduling

Moving-head Disk Mechanism



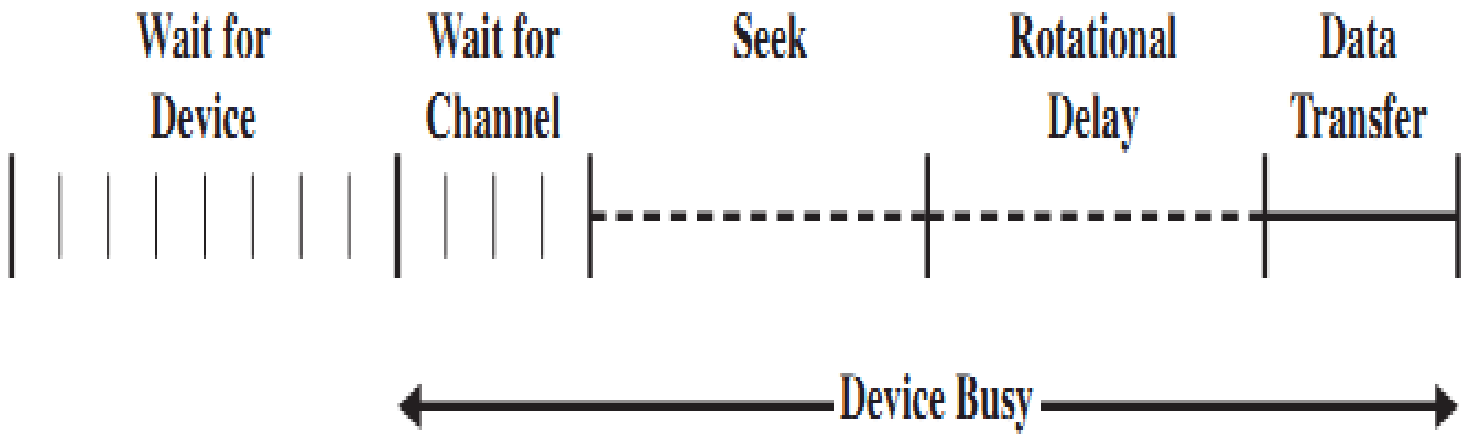
Elements of File Management



Disk Scheduling (1)

- The operating system is responsible for using hardware efficiently — for the disk drives, this means having a fast access time and disk bandwidth.
- Access time has two major components:
  - Seek time is the time for the disk are to move the heads to the cylinder containing the desired sector.
  - Rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head.
- Minimize seek time ≈ seek distance.
- Disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of last transfer.

Components of Disk I/O Transfer



ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Disk Scheduling (2)

- There are many sources of disk I/O request:
  - OS
  - System processes
  - Users processes
- I/O request includes input/output mode, disk address, memory address, number of sectors to transfer.
- OS maintains queue of requests, per disk or device.
- Idle disk can immediately work on I/O request, busy disk means work must queue:
  - Optimization algorithms only make sense when a queue exists.

Disk Structure

- Disk drives are addressed as large 1-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer.
- The 1-dimensional array of logical blocks is mapped into the sectors of the disk sequentially:
  - Sector 0 is the first sector of the first track on the outermost cylinder.
  - Mapping proceeds in order through that track, then the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

Disk Scheduling Algorithms

- Note that drive controllers have small buffers and can manage a queue of I/O requests (of varying “depth”).
- Several algorithms exist to schedule the servicing of disk I/O requests.
- The analysis is true for one or many platters.
- We illustrate them with a I/O request queue (cylinders are between 0-199):

  queue = 98, 183, 37, 122, 14, 124, 65, 67

  head starts at 53

First Come First Serve (FCFS) Example

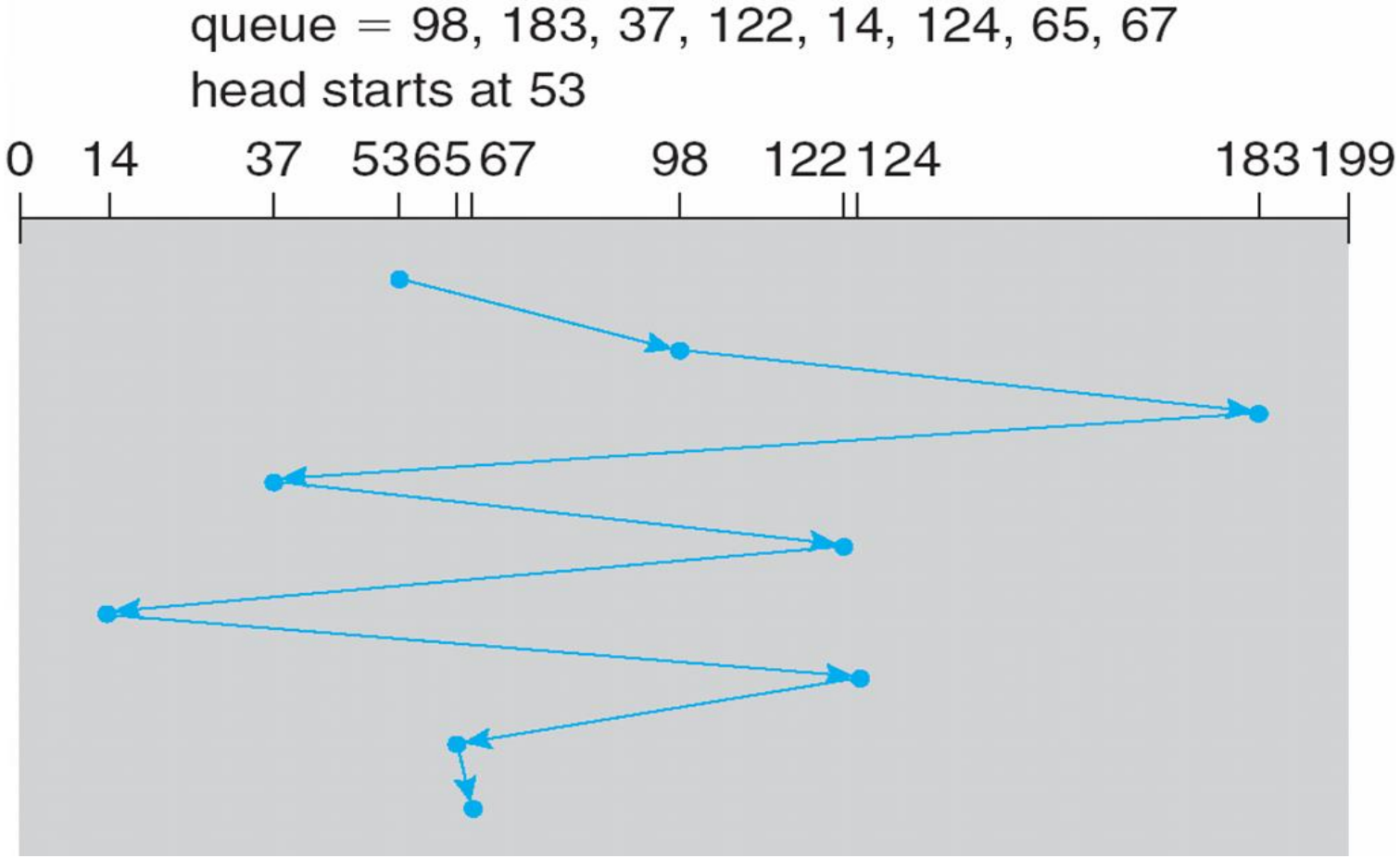


Illustration shows total head movement of 640 cylinders.

First Come First Serve (FCFS)

- Handle I/O requests sequentially.
- Fair to all processes.
- Approaches random scheduling in performance if there are many processes/requests.
- Suffers from global zigzag effect.

Shortest Seek Time First (SSTF) Example

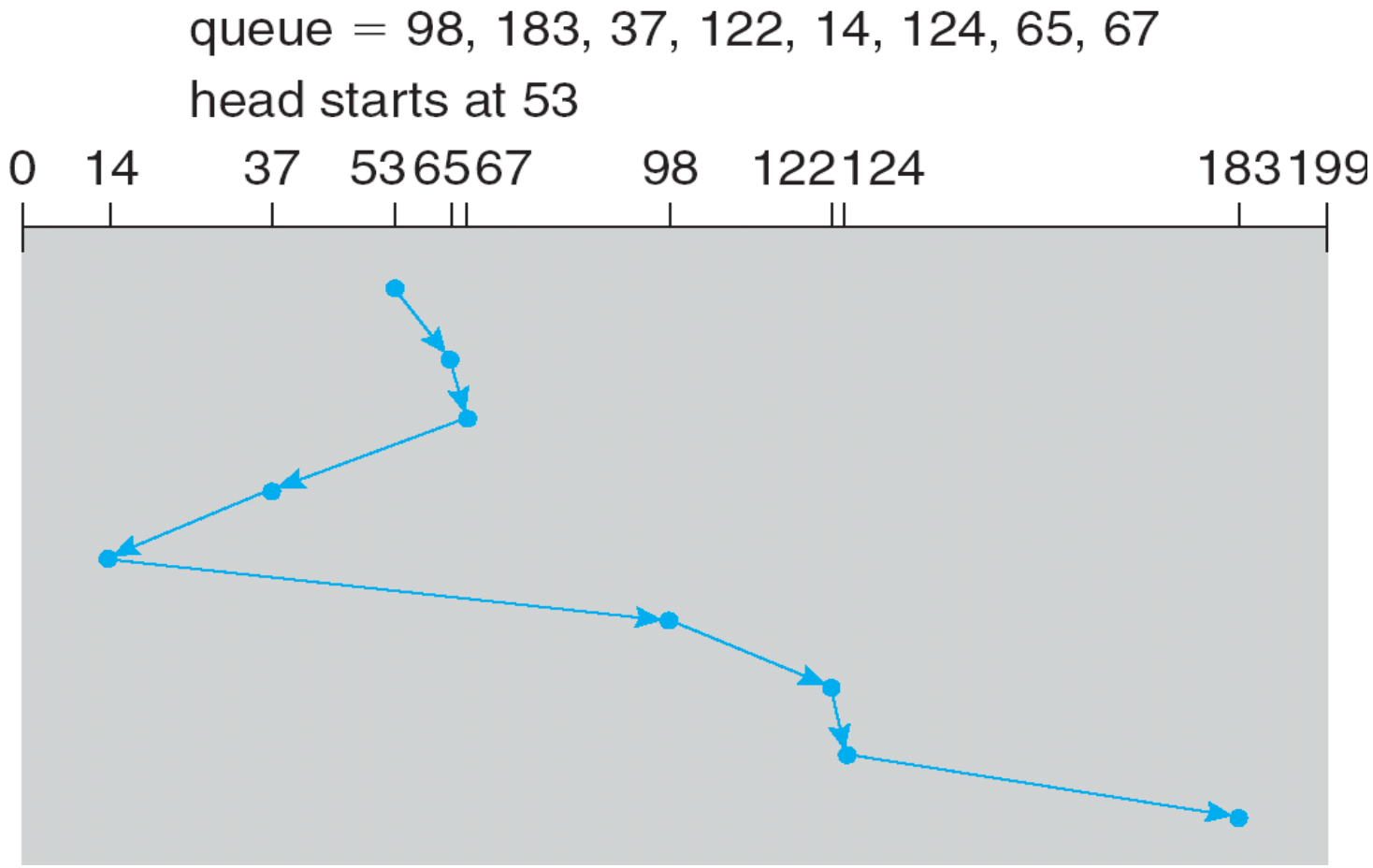


Illustration shows total head movement of 236 cylinders.

Shortest Seek Time First (SSTF)

- Selects the request with the minimum seek time from the current head position.
- Also called Shortest Seek Distance First (SSDF) – It's easier to compute distances.
- It's biased in favor of the middle cylinders requests.
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.

Elevator Algorithms

- Algorithms based on the common elevator principle.
- Four combinations of Elevator algorithms:
  - Service in both directions or in only one direction.
  - Go until last cylinder or until last I/O request.

Direction \ Go until	Go until the last cylinder	Go until the last request
Service both directions	**Scan**	**Look**
Service in only one direction	**C-Scan**	**C-Look**

Scan Example

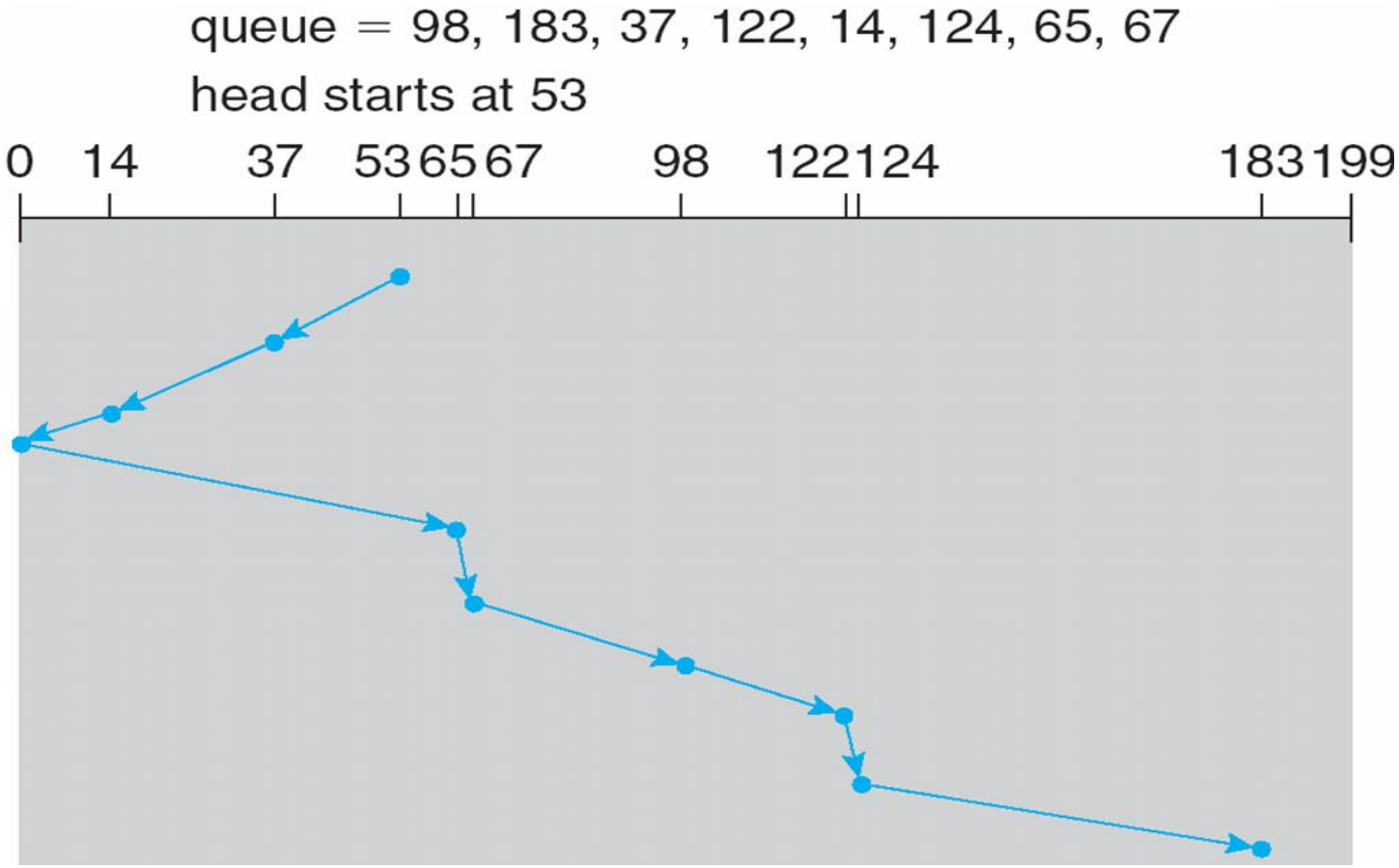


Illustration shows total head movement of 208 cylinders.

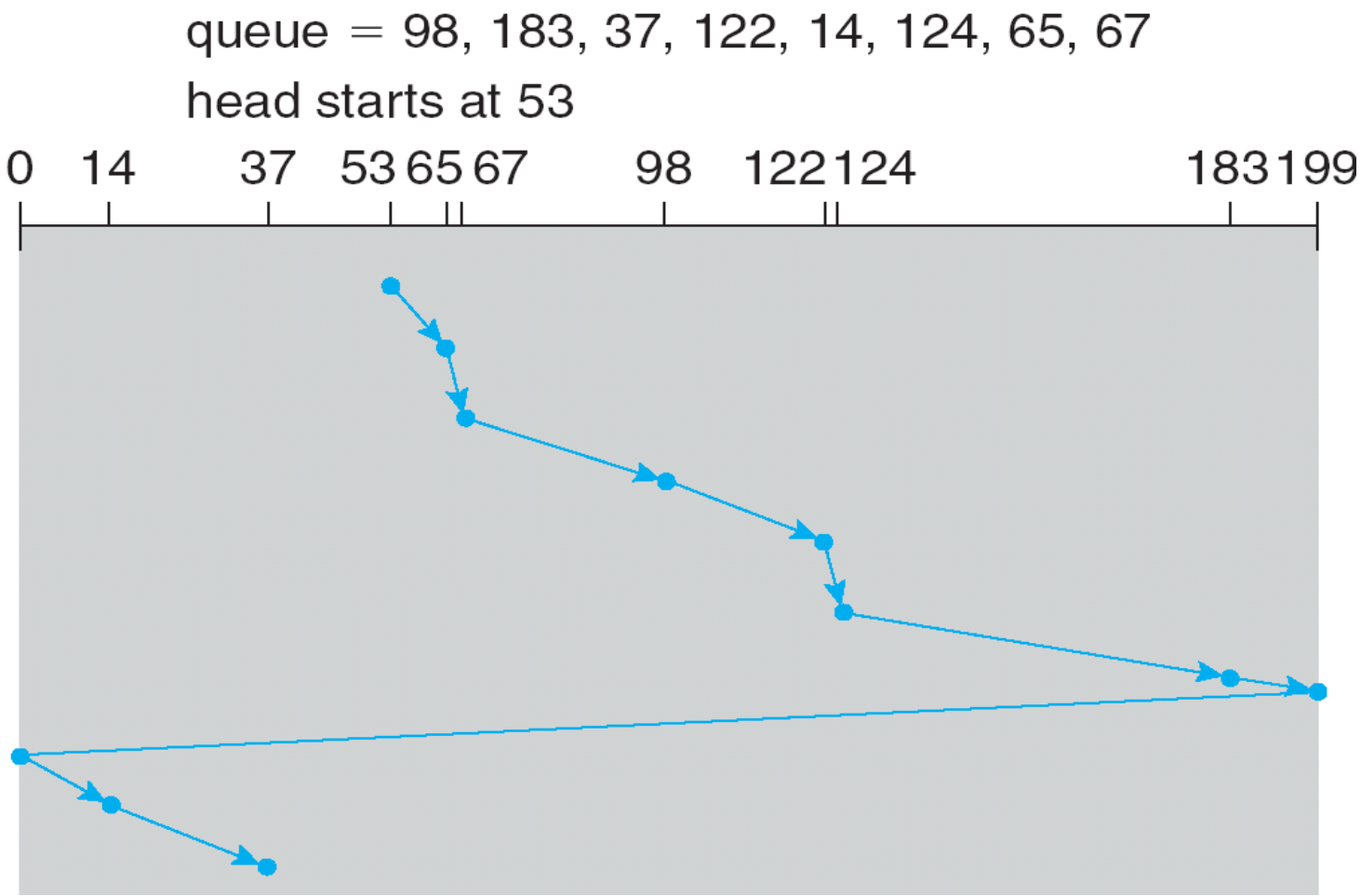
Scan

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- It moves in both directions until both ends.
- Tends to stay more at the ends so more fair to the extreme cylinder requests.

Look

- The disk arm starts at the first I/O request on the disk, and moves toward the last I/O request on the other end, servicing requests until it gets to the other extreme I/O request on the disk, where the head movement is reversed and servicing continues.
- It moves in both directions until both last I/O requests; more inclined to serve the middle cylinder requests.

C-Scan Example

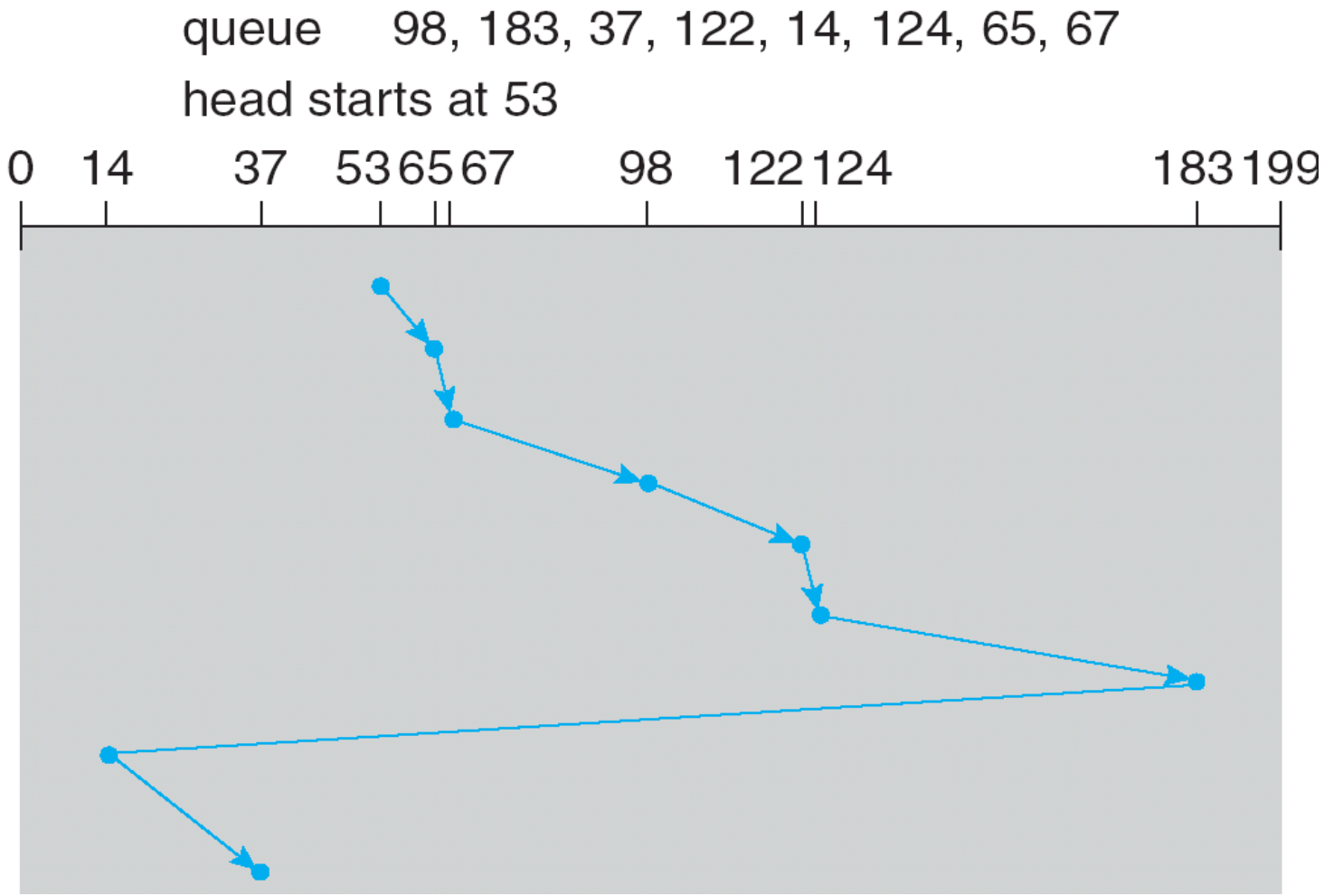


ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

C-Scan

- The head moves from one end of the disk to the other, servicing requests as it goes. When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one.
- Provides a more uniform wait time than SCAN; it treats all cylinders in the same manner.

C-Look Example



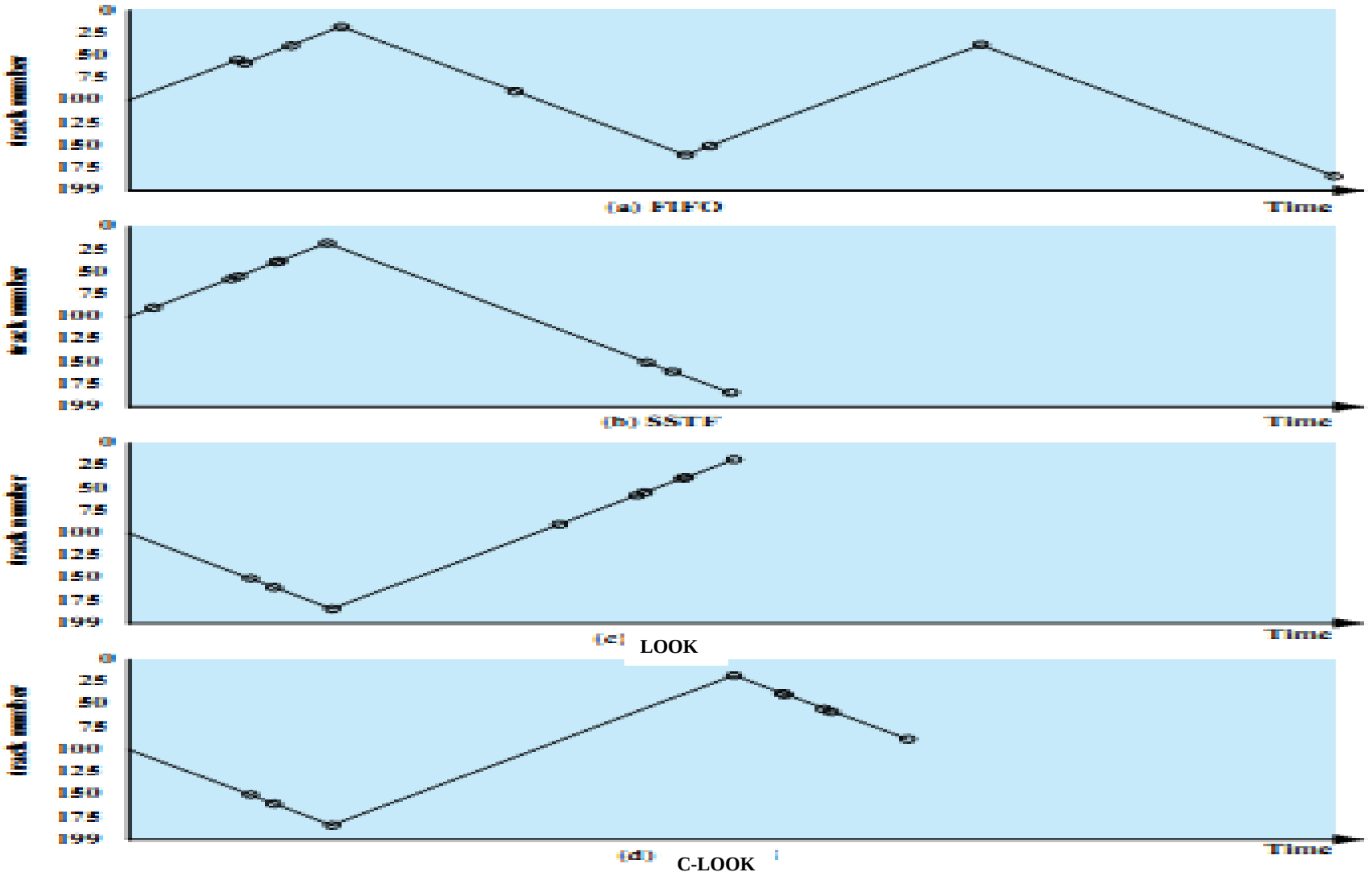
C-Look

- Look version of C-Scan.
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk.
- In general, Circular versions are more fair but pay with a larger total seek time.
- Scan versions have a larger total seek time than the corresponding Look versions.

Another Example

(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) LOOK (starting at track 100, in the direction of increasing track number)		(d) C-LOOK (starting at track 100, in the direction of increasing track number)	
Next track accessed	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	**Average seek length**	27.5	**Average seek length**	27.8	**Average seek length**	35.8

Graphs for previous example



Other Disk Scheduling Policies

- Pickup
  - A combination of FCFS and Look.
  - Goes to next I/O request by FCFS but services all existing requests on the way to it.
- Priority
  - Goal is not to optimize disk use but to meet other objectives.
  - Short batch jobs may have higher priority.
  - Provide good interactive response time.

Scan Algorithm Variations

- FScan
  - Use two queues.
  - One queue is empty to receive new requests.
- N-step-Scan
  - Segments the disk request queue into subqueues of length N.
  - Subqueues are processed one at a time, using Scan.
  - New requests added to other queue when a certain queue is processed.

ie
SCHOOL OF HUMAN SCIENCES & TECHNOLOGY

Selecting a Disk-Scheduling Algorithm (1)

- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file-allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating system, allowing it to be replaced with a different algorithm if necessary.

Selecting a Disk-Scheduling Algorithm (2)

- With low load on the disk, It's FCFS anyway.
- SSTF is common and has a natural appeal – good for medium disk load.
  - SCAN and C-SCAN perform better for systems that place a heavy load on the disk; Less starvation.
  - Performance depends on number and types of requests.
- Requests for disk service can be influenced by the file-allocation method and metadata layout.
- Either SSTF or LOOK (as part of an Elevator package) is a reasonable choice for the default algorithm.

SCHOOL OF
HUMAN SCIENCES
& TECHNOLOGY