CS3103 Cheatsheet Final Exam

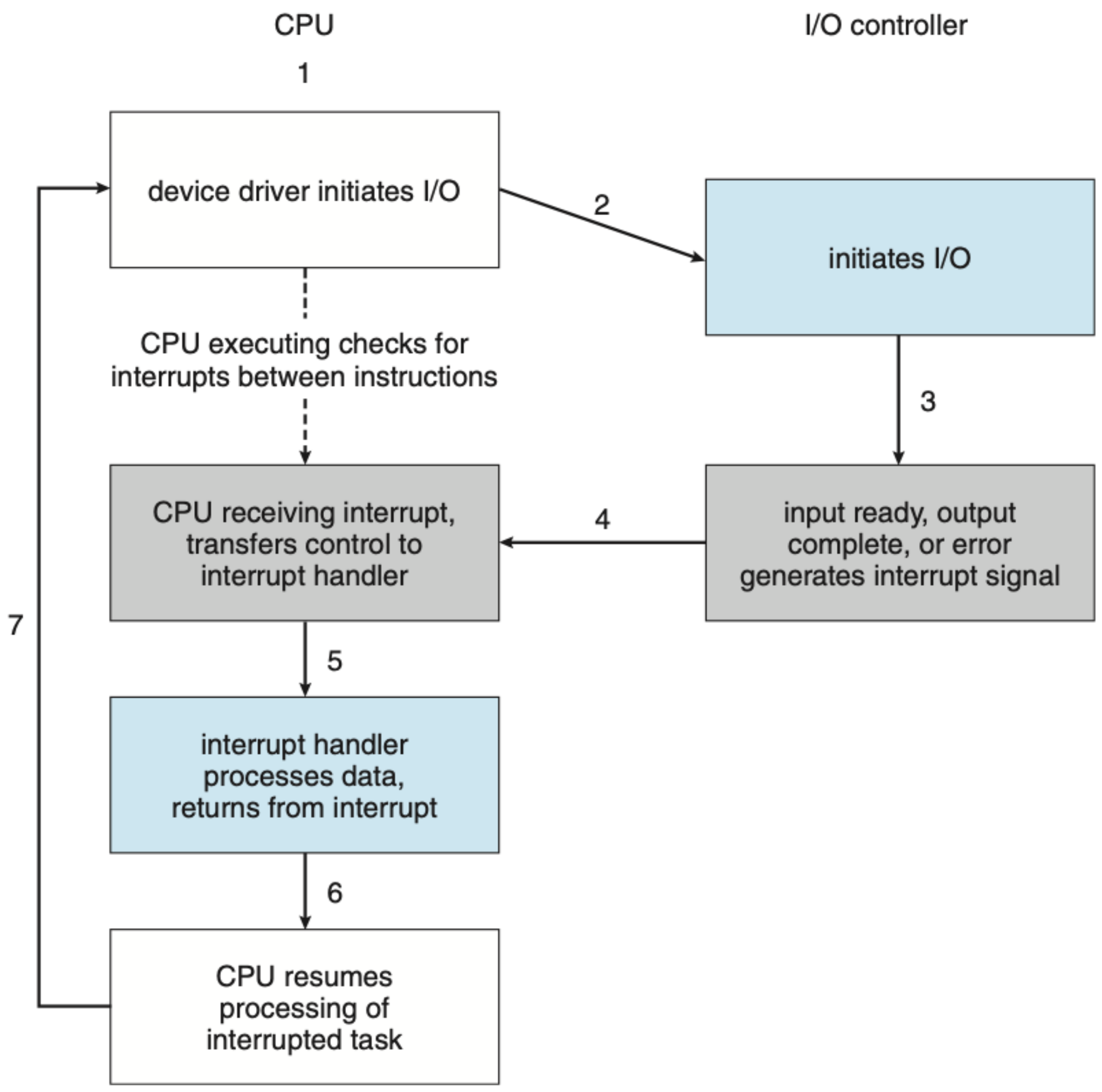
Introduction

Main Purpose of Operating System

1. To provide an environment for a computer user to execute programs on computer hardware in a convenient and efficient manner.
2. To allocate the separate resources of the computer as needed to solve the problem given. The allocation process should be as fair and efficient as possible.
3. As a control program it serves two major functions:
    1. Supervision of the execution of user programs to prevent errors and improper use of the computer, and
    2. Management of the operations and control of I/O devices

Interrupt Service Routine

1. Device controller informs CPU that it has finished its operation by causing an interrupt
   1. Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the address of all the service routines.
   2. Interrupt architecture must save the address of the interrupted instruction.
   3. A `trap` or `exception` is a software-generated interrupt caused either by an error or a user request.
   4. Operating system is interrupt driven.

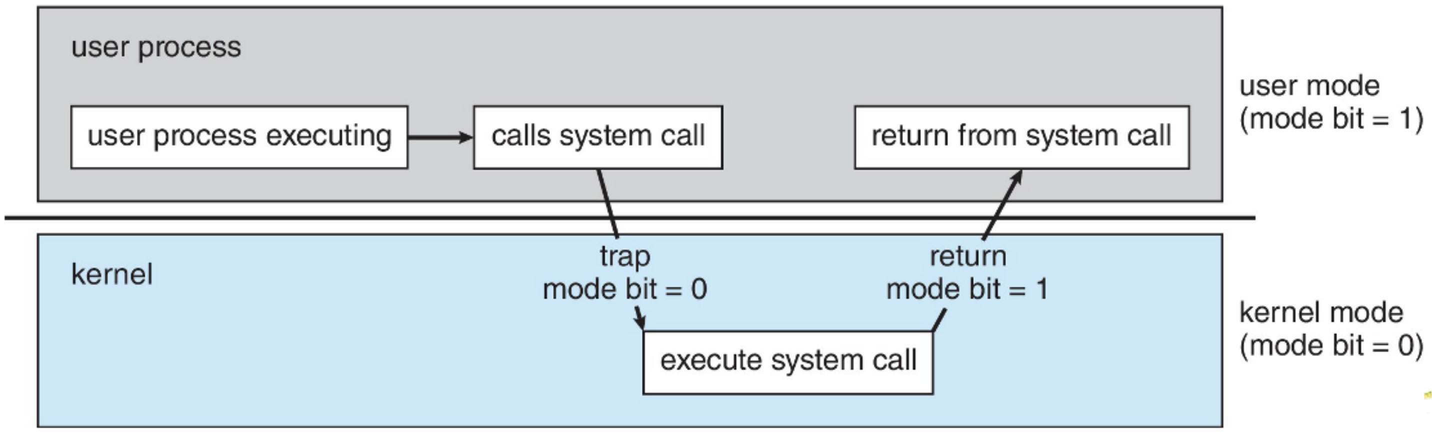

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task
7

Dual-mode Operation

Dual-mode operation allows OS to protect itself and other system components, as there are user mode and kernel mode.

Mode bit is provided by hardware to distinguish when system is running user code or kernel code,

1. When user code is running, then mode bit is `user`.
2. When kernel code is executing, then mode bit is `kernel`.
   Then system call changes mode to kernel, return from call resets it to user.



OS Structure

System vs. API

1. System calls are detailed and more difficult to work with.
2. Using APIs can allow program to compile and run on other system.

Microkernels

1. Moves as much from the kernel into user space.
2. Communication between user modules using message passing.
3. Benefits:
   1. Easier to extend a microkernel.
   2. Easier to port the OS to new architectures.
   3. More reliable (less code is running in kernel code); more secure.
4. Detriments:
   1. Performance overhead of user space to kernel space communication.

Processes

Process Concept

Process is a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process.

It contains multiple parts:

1. Stack containing temporary data, e.g., function parameters, return addresses, local variables, and etc.
2. Current activity including program counter, processor registers.
3. Heap containing memory dynamically allocated during runtime.

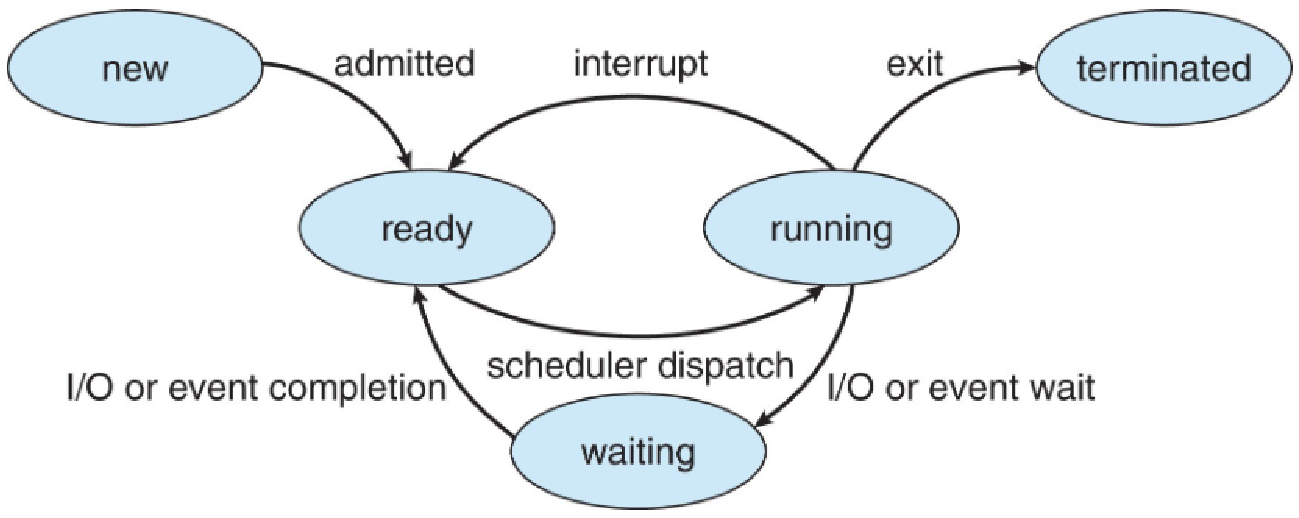
4. Data section containing global variables.
5. The program code, also called text section.

Program is passive entity stored on disk (executable file); process is active, and program becomes process when an executable file is loaded into memory.

Process State

As a process executes, it changes state:

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



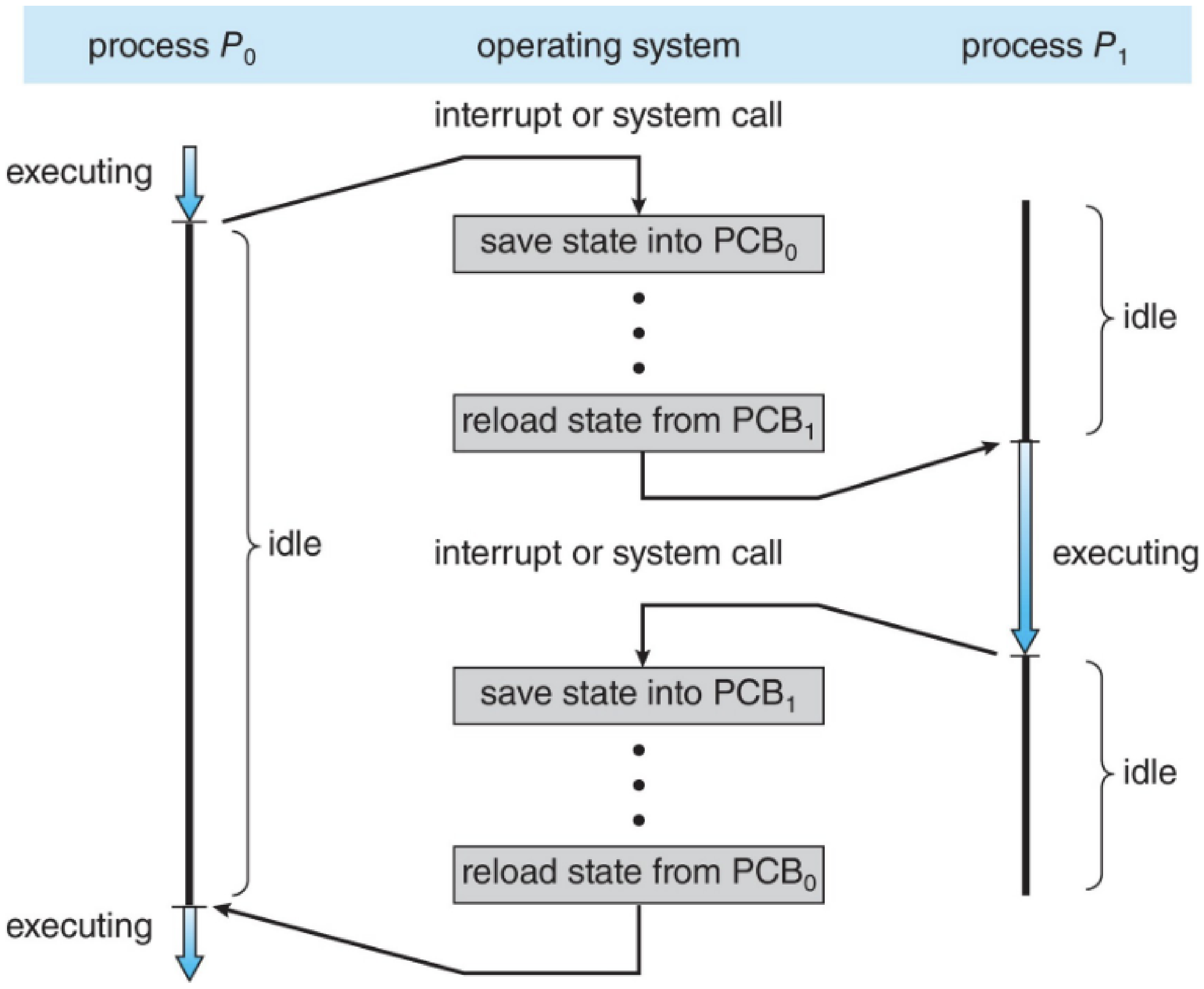
Process Control Block (PCB)

Information associated with each process (also called task control block),

1. Process state: running, waiting, etc.
2. Program counter: location of instruction to next execute.
3. CPU registers: contents of all process-centric registers.
4. CPU scheduling information: priorities, scheduling queue pointers.
5. Memory management information: memory allocated to the process.
6. Accounting information: CPU used, clock time, elapsed since start, time limits.
7. I/O status information: I/O devices allocated to process, list of open files.

Context Switch (from Process to process)

A context switch occurs when the CPU switches from one process to another.



When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch.

Context of a process is represented in the PCB.

Context-switch time is pure overhead; the system does no useful work while switching, the more complex the OS and the PCB causes longer time on context switch.

Time dependent on hardware support, as some hardware provides multiple sets of registers per CPU, then multiple contexts is loaded at once.

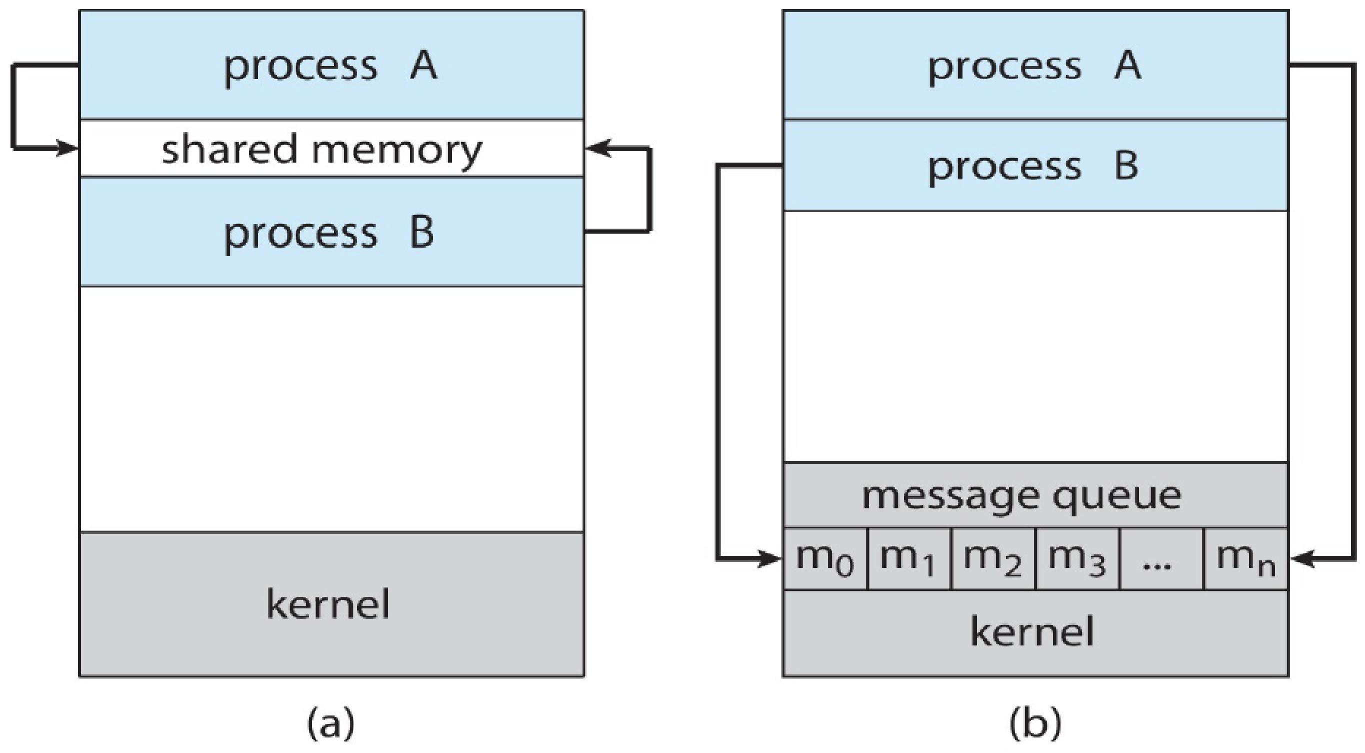
Interprocess Communication (IPC)

Cooperating processes need IPC, and there are two models of IPC:

1. Shared memory
    1. It needs to setup shared area among the process that wish to communicate first.
    2. It is more suitable for large amount of data.
    3. The communication is under the control of the users processes, not OS.
2. Message passing
    1. It is efficient for small size data.
    2. Processes communicate with each other without resorting to shared variables, `send` and `receive`

3. If processes wish to communicate, they need to establish communication link between them, as exchange messages via `send/receive`

(a) Shared memory. (b) Message passing.



Producers and Consumer Problem

There is a paradigm for cooperating processes: producer process produces information that is consumed by a consumer process.

There are two variations:

1. Unbounded-buffer places no practical limit on the size of the buffer:
   1. Producer never waits.
   2. Consumer waits if there is no buffer to consume.
2. Bounded-buffer assumes that there is a fixed buffer size:
   1. Producer must wait if all buffers are full.
   2. Consumer waits if there is no buffer to consume.

Direct Communication and Indirect Communication

In direction communication, processes must name each other explicitly:

1. `send(P, message)`: send a message to process `P`.
2. `receive(Q, message)`: receive a message from process `Q`.

There are some properties of the communication link:

1. Links are established automatically.
2. A link is associated with exactly one pair of communicating processes.

3. Between each pair there exists exactly one link.
4. The link may be unidirectional, but it is usually bidirectional.

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

1. Each mailbox has a unique id.
2. Processes can communicate only if they share a mailbox

There are some properties of the communication link:

1. Link established only if processes share a common mail.
2. A link may be associated with many processes.
3. Each pair of processes may share serval communication links.
4. Link may be unidirectional or bidirectional.

There are some common operations:

1. Create a new mailbox (port).
2. Send and receive messages through mailbox.
3. Delete a mailbox.

Primitives are defined as:

1. `send(A, message)`: to send a message to mail `A`.
2. `receive(A, message)`: to receive a message from mail `A`.

Mailbox sharing:

1. P_1, P_2 and P_3 share mailbox `A`.
2. P_1 sends; P_2 and P_3 receive.

Threads & Concurrency

Four Benefits of using Threads

1. Responsiveness: may allow continued execution if part of process is blocked, especially important for user interfaces.
2. Resource sharing: threads share resources of process, easier than shared memory or message passing.
3. Economy: cheaper than process creation, thread switching lower overhead than context switching.
4. Scalability: process can take advantage of multicore architectures.

Mapping between User Thread and Kernel Thread

User threads is managed by user-level threads library, there are three primary thread libraries:

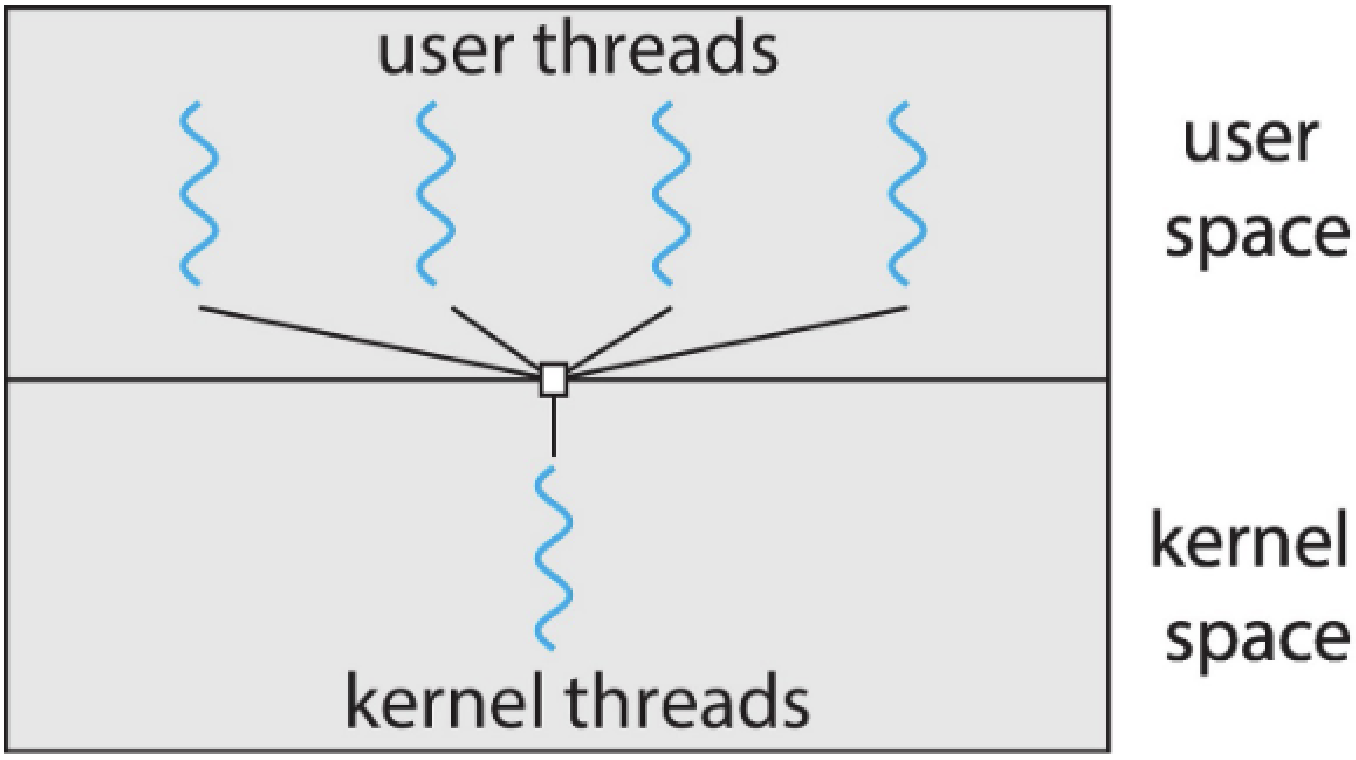
1. POSIX Pthreads.
2. Windows threads.
3. Java threads.

Kernel threads is supported by the kernel.

There are four mapping model between user threads and kernel threads:

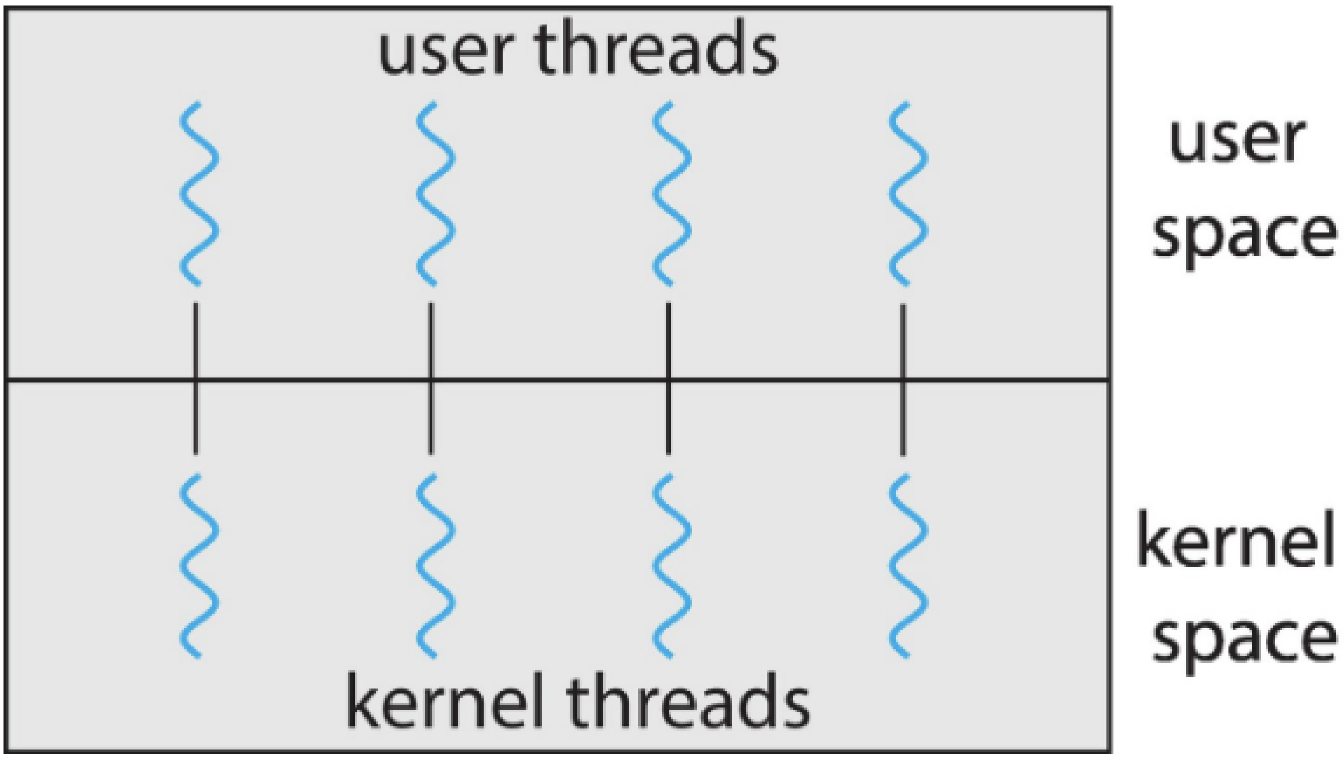
Many-to-one: many user-level threads mapped to single kernel thread.

1. One thread blocking causes all to block.
2. Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time.



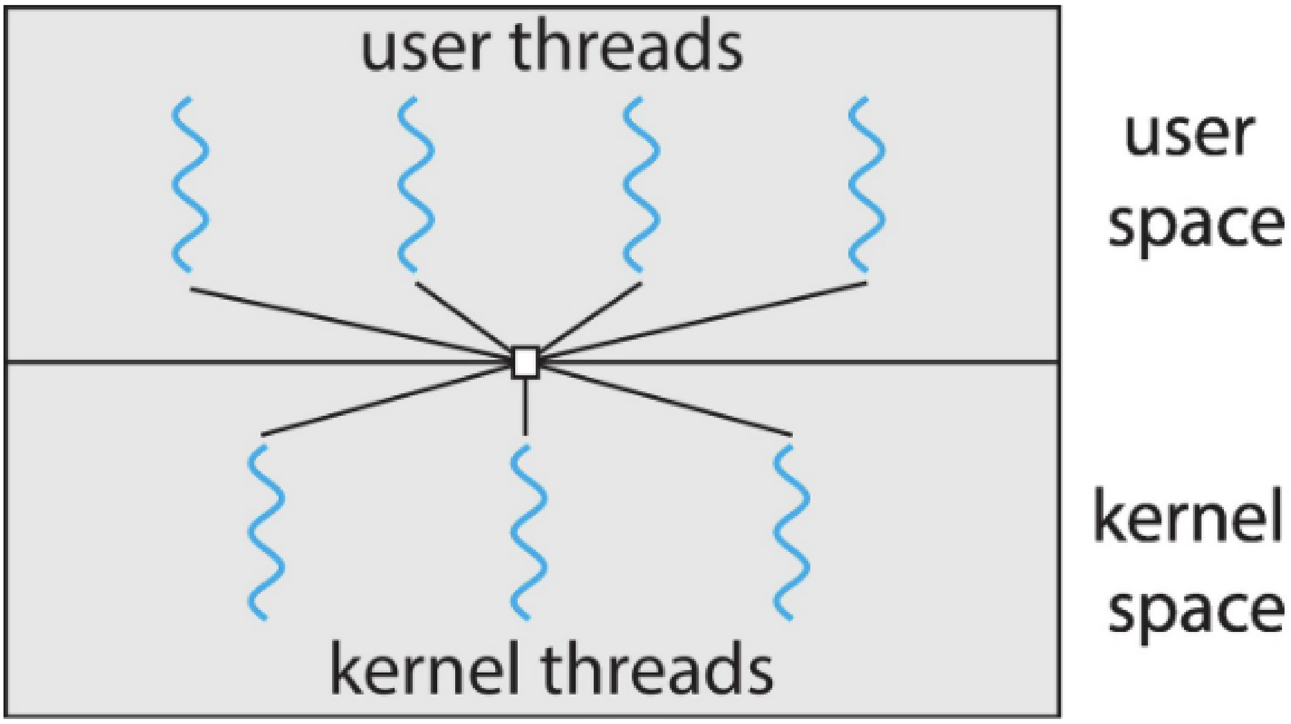
One-to-one: each user-level thread maps to kernel thread.

1. Creating a user-level thread creates a kernel thread.
2. More concurrency than many-to-one.
3. Number of threads per process sometimes restricted due to overhead.

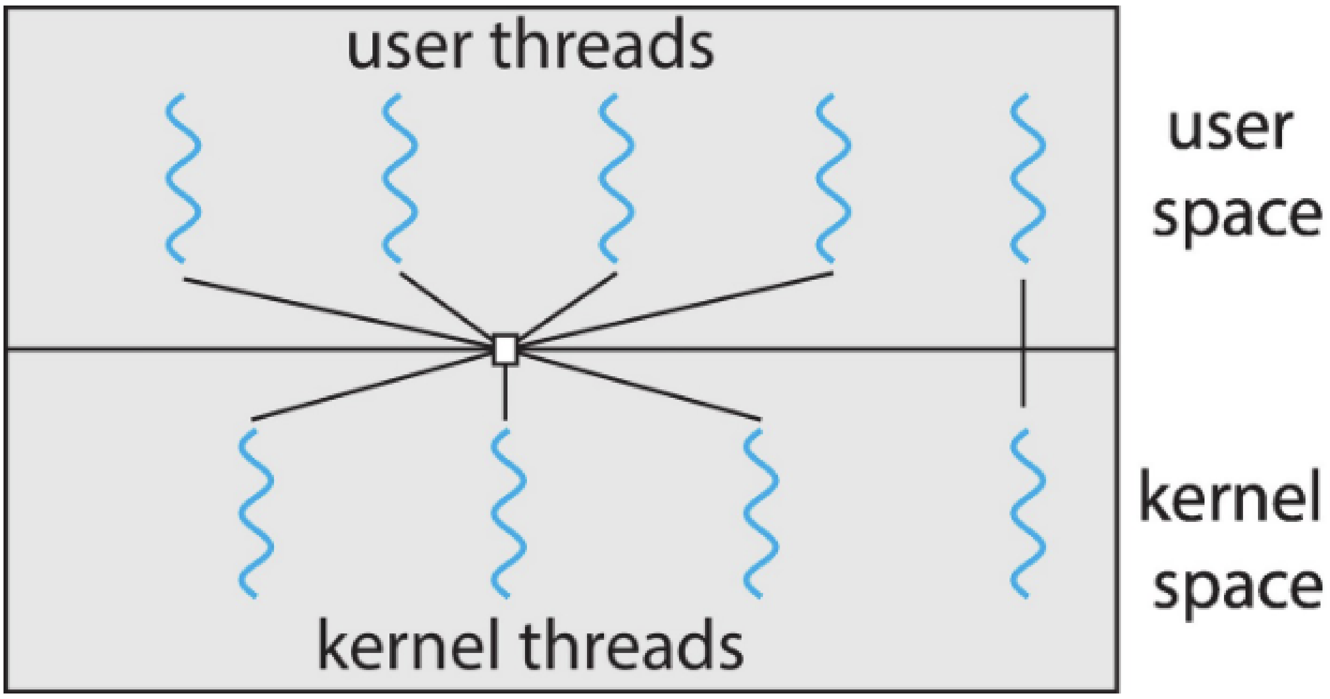


Many-to-many model: allows many user level threads to be mapped to many kernel threads.

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



Two-level model: it is similar to `m:m`, except that it allows a user thread to be bound to kernel thread.



Threads Pools

It creates a number of threads in a pool where they await work.

There are many advantages:

1. Usually slight faster to service a request with an existing thread than create a new thread.
2. Allows the number of threads in the application to be bound to the size of the pool.
3. Separating task to be performed from mechanics of creating task allows different strategies for running task, i.e., tasks could be scheduled to run periodically.

CPU Scheduling

Preemptive vs. Non-preemptive

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.

When scheduling takes place only under circumstance 1 and 4, the scheduling scheme is nonpreemptive, otherwise, it is preemptive.

Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until the process is released, either by terminating or by switching to the waiting state.

Preemptive scheduling can result in race conditions when data are shared among several processes.

Scheduling Criteria

1. CPU utilisation: keep the CPU as busy as possible.
2. Throughput: the number of processes that complete their execution per time unit.
3. Turnaround time: amount of time to execute a particular process.
    1. $\text{turnaround time} = \text{finish time - arrival time} = \text{waiting time} + \text{CPU time(burst time)}.$
4. Waiting: amount of time of a process has been waiting in the ready queue.
5. Response time: amount of time that it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment and it is the first segment of waiting time).

Therefore, to optimise:

1. Max CPU utilisation.
2. Max throughput.
3. Min turnaround time.
4. Min waiting time.
5. Min response time.

FCFS

It is simply first-come, first-served scheduling.

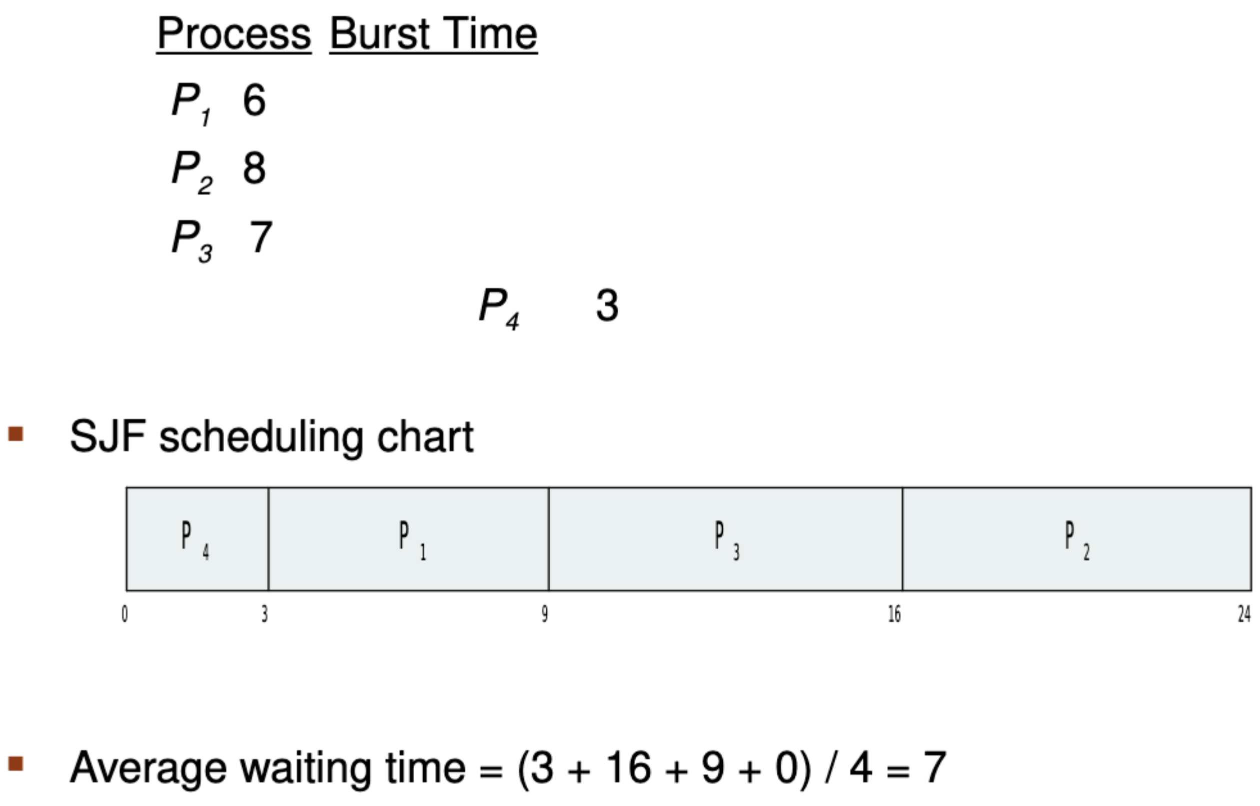
Convoy effect: short process behind long process.

SJF

Associate with each process the length of its next CPU burst, then use these lengths to schedule the process with the shortest time.

It is optimal, i.e., gives minimum average waiting time for a given set of processes.

Non-preemptive version:

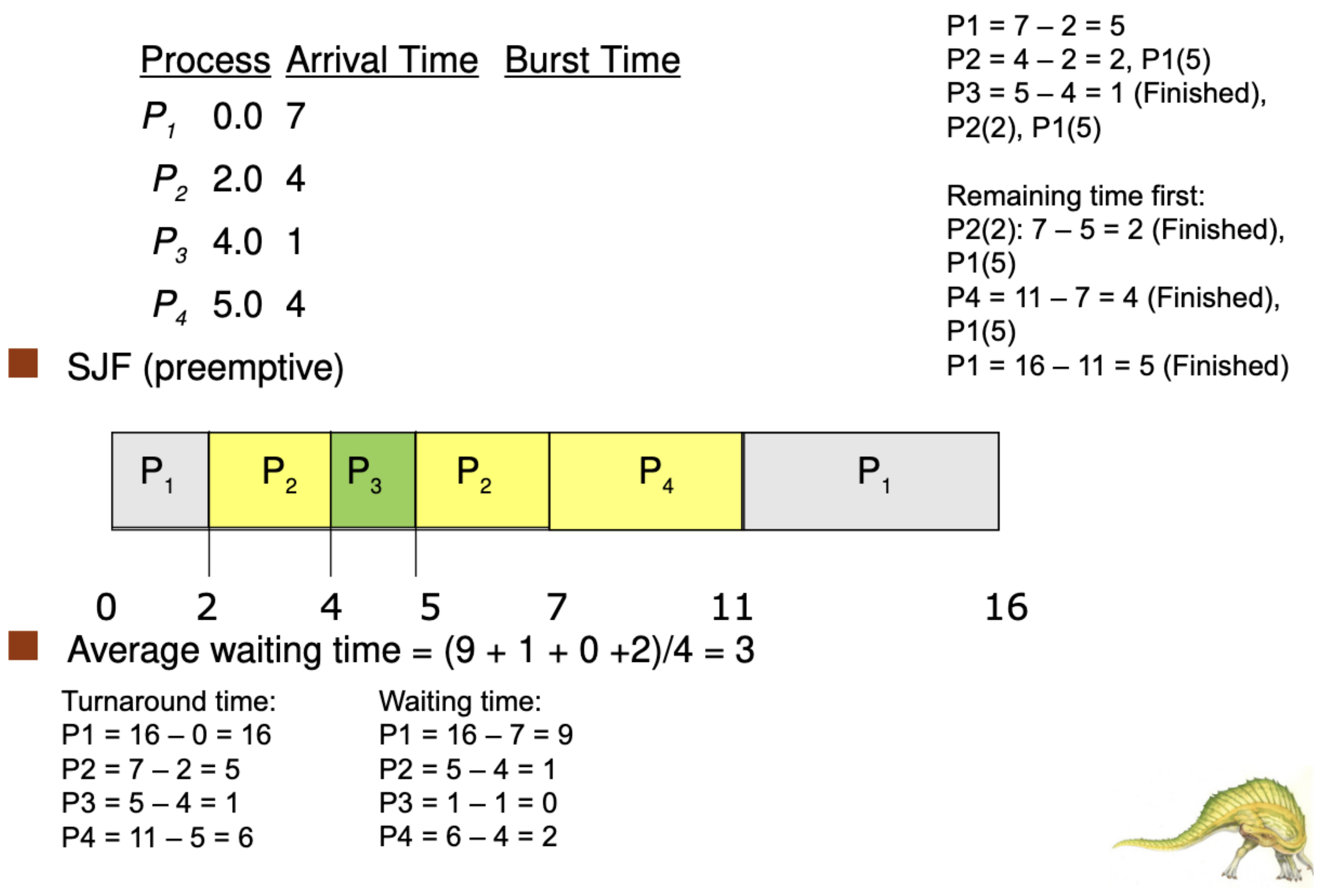


Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart

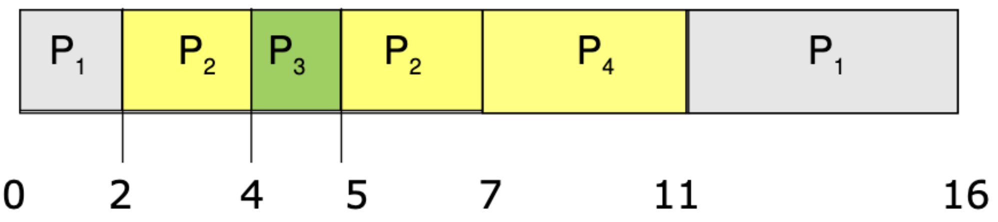
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Preemptive version:



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 (preemptive)



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

Turnaround time:	Waiting time:
P1 = 16 – 0 = 16	P1 = 16 – 7 = 9
P2 = 7 – 2 = 5	P2 = 5 – 4 = 1
P3 = 5 – 4 = 1	P3 = 1 – 1 = 0
P4 = 11 – 5 = 6	P4 = 6 – 4 = 2

P1 = 7 – 2 = 5
P2 = 4 – 2 = 2, P1(5)
P3 = 5 – 4 = 1 (Finished), P2(2), P1(5)

Remaining time first:
P2(2): 7 – 5 = 2 (Finished), P1(5)
P4 = 11 – 7 = 4 (Finished), P1(5)
P1 = 16 – 11 = 5 (Finished)

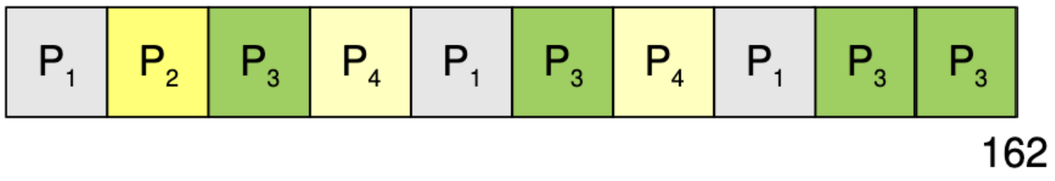
RR

Each process gets a small unit of CPU time (time quantum q), 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 units at once. No process waits more than $(n-1)q$ time units.

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



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

Time Quantum q

1. If q is extremely large, then RR is the same as the FCFS.
2. If q is small, then q must be large w.r.t. context switch, otherwise overhead is too high.
3. q should be large compared to context switch time, as $10 \text{ ms} \leq q \leq 100 \text{ ms}$, context switch < 10 microseconds.

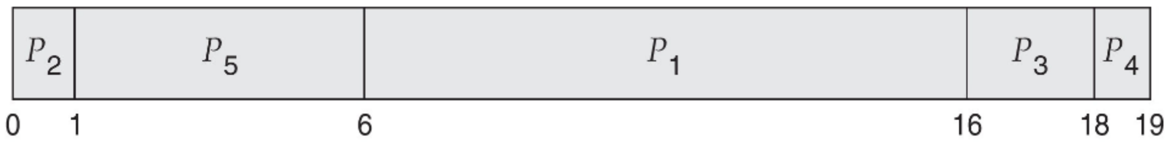
Priority Scheduling

A priority number is associated with each process with the highest priority (smallest integer → highest priority)

Starvation problem: low priority processes may never execute. With a solution called aging: as time progresses increase the priority of the process.

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

- Priority scheduling Gantt Chart



Multilevel Queue

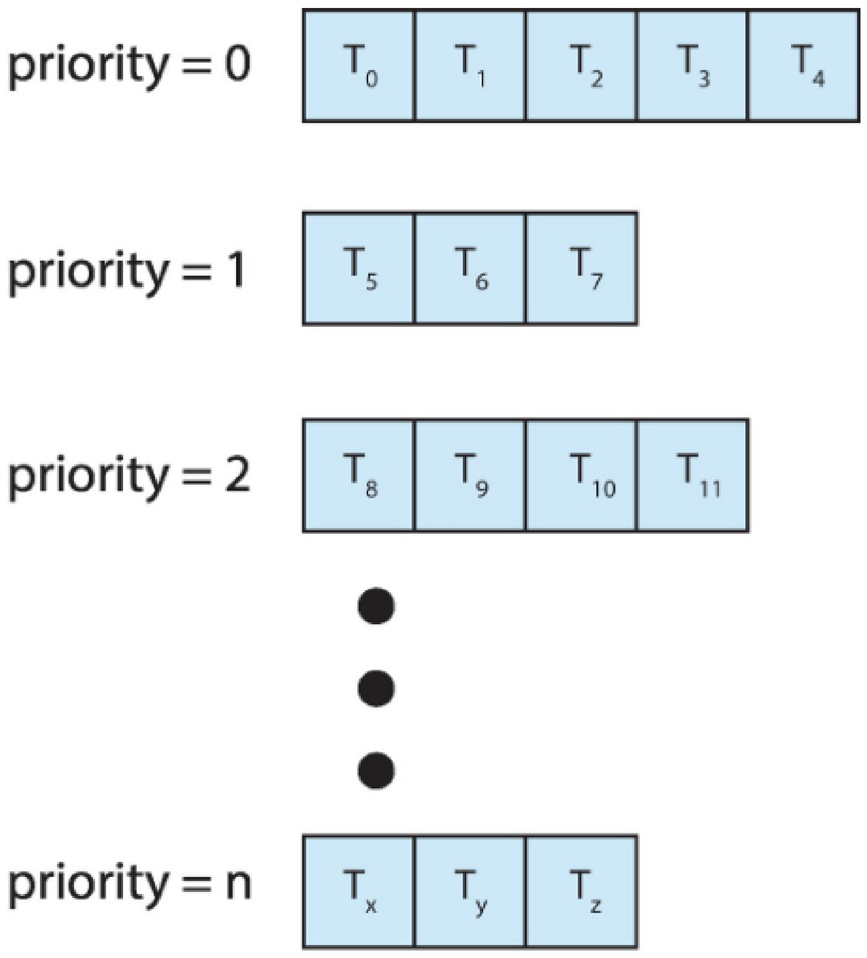
Ready queue is partitioned into separate queues:

1. Foreground (interactive): RR.
2. Background (batch): FCFS.

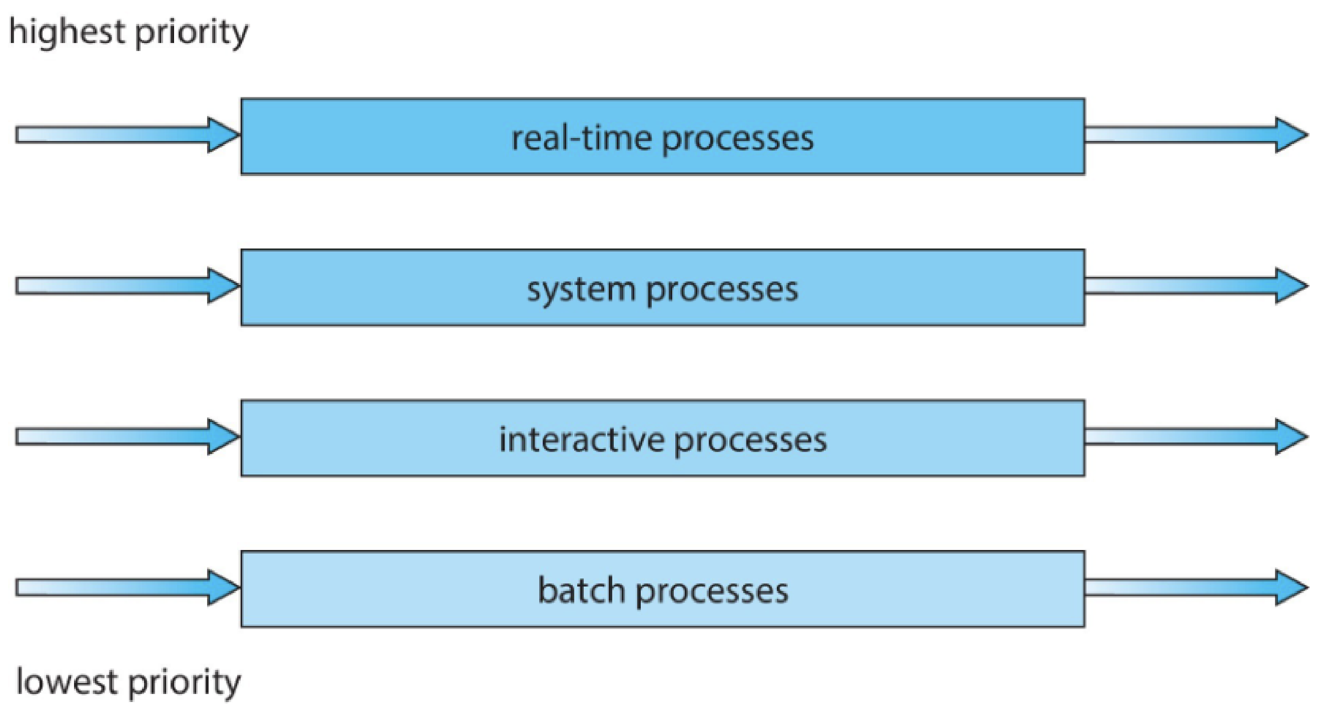
Scheduling must be done between the queues:

1. Fixed priority scheduling, i.e., serve all from foreground, then from background).
2. Time slice, i.e., 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 the background in FCFS.

With priority scheduling, they have separate queues for each priority, i.e.,



Prioritisation based upon process type, i.e.,



Synchronisation Tools

Producer and Consumer Problem

Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Race Condition

Producer:

```
while (true) {
// produce an item and put in nextProduced
        while(count == BUFFER_SIZE); // do nothing
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

Consumer:

```
while (true) {
        while (count == 0); // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        // consume the item in nextConsumed
}
```

`count++` could be implemented as:

```
register1 = count;
register1 = register1 + 1;
count = register1;
```

`count--` could be implemented as:

```
register2 = count;
register2 = register2 - 1;
count = register2;
```

Consider this execution interleaving with `count = 5` initially:

1. S0: producer execute `register1 = count`; (`register1 = 5`).
2. S1: producer execute `register1 = register1 + 1`; (`register1 = 6`).
3. S2: consumer execute `register2 = count`; (`register2 = 5`).
4. S3: consumer execute `register2 = register2 - 1`; (`register2 = 4`).
5. S4: producer execute `count = register1`; (`count = 6`).
6. S5: consumer execute `count = register2`; (`count = 4`).

Critical Section Problem

Consider system of n processes $\{p_0, p_1, \ldots, p_{n-1}\}$, each process has critical section segment of code, i.e.,

1. Process may be changing common variables, updating table, writing file, etc.
2. When one process in critical section, no other may be in its critical section.

Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section.

There are some solutions to critical section problem:

1. Mutual exclusion: if process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
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 processes that will enter the critical section next cannot be postponed indefinitely.
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.
    1. Assume that each process executes at a nonzero speed.
    2. No assumption concerning relative speed of the N processes.

Atomic Instruction

Many system provide hardware support for critical section code.

Uniprocessors: could disable interrupts, i.e.,

1. Currently running code would execute without preemption.
2. Generally too inefficient on multiprocessor systems,
    1. OS using this not broadly scalable.

Atomic instruction means that it is non-interruptable,

1. Either test memory word and set value → `testandset()`, or `swap()`.

`testandset()`

Its definition is:

```
boolean testandset (boolean *target) {
        boolean rv = *target;
        *target = TRUE;
        return rv;
}
```

If `*target = TRUE`, then `testandset = TRUE`; `*target = TRUE`.

If `*target = FALSE`, then `testandset = false`; `*target = TRUE`.

Suppose shared boolean variable `lock`, initialised to `false`:

```
do {
        while (testandset(&lock)); // do nothing
        // critical section
        lock = False;
        // remainder section
} while (true);
```

Bounded waiting with `testandset`, suppose shared boolean variable `lock`, initialised to `false`, and the solution for i process (total n number of processes):

```
do {
        waiting[i] = True;
        while (waiting[i] && testandset(&lock)); // do nothing
        waiting[i] = False;
        // critical section
        j = (i + 1) % n;
        while ((j≠1 && !waiting[j]))
                j = (j + 1) % n;
        if (j==i)
                lock = False;
        else
                waiting[j] = False;
        // remainder section
}
```

Swap()

Its definition is:

```
void Swap(boolean *a, boolean *b) {
        boolean temp = *a;
        *a = *b;
        *b = temp;
}
```

Suppose shared boolean variable `lock` initialised to `False`, and each process has a local boolean variable key.

```
do {
        key = True;
        while (key == True)
                swap (&lock, &key);
        // critical section
        lock = False;
        // remainder section
} while (True);
```

Semaphore

Suppose semaphore `S` as a integer variable, there are two standard operation modify `S`, i.e., `wait()` and `signal()`.

It can only be accessed via two indivisible (atomic) operations:

```
wait(S) {
        while S <= 0; // do nothing
        S--;
}

signal(S) {
        S++;
}
```

Binary semaphore is that the integer value can range only between 0 and 1, which is simpler to implement, i.e., mutex locks:

```
semaphore S; // initialised to 1
wait(S);
// critical section
signal(S);
```

It must guarantees that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time.

Semaphore implementation with no busy waiting is that with each semaphore, there is an associated waiting queue and each entry in a waiting queue has two data items:

1. Value of type integer.
2. Pointer to next record in the list.

There are two operations:

1. `block()`: place the process invoking the operation on the appropriate waiting queue.
2. `waitup()`: remove one of processes in the waiting queue and place it in the ready queue.

```
wait (S) {
        value--;
        if (value < 0) {
                // add this process to waiting queue
                block();
        }
}
```

```
signal (S) {
        value++;
        if (value ⩾ 0) {
                // remove a process P from the waiting queue
                wakeup(P);
        }
}
```

Bounded Buffer Problem

Suppose there is N buffers, each can hold one item. Semaphore `mutex` is initialised to the 1, semaphore `full` is initialised to the 0, and semaphore empty is initialised to N.

The structure of the producer process:

```
do {
        // produce an item
        wait(empty);
        wait(mutex);
        // add the item to the buffer
        signal(mutex);
        signal(full);
} while (True);
```

The structure of the consumer process:

```
do {
        wait(full);
        wait(mutex);
        // remove an item from buffer
        signal(mutex);
        signal(empty);
        // consume the removed item
} while (True);
```

Readers and Writers Problem

A data set is shared among a number of concurrent processes:

1. Readers: only read the data set, and they do not perform any updates.
2. Writers: can both read and write.

The problem is to allow multiple readers to read at the same time but only one single writer can access the shared data at the same time.

Suppose there is a data set, semaphore `mutex` is initialised to 1, semaphore `wrt` is initialised to 1, and integer `readcount` is initialised to 0.

The structure of a writer process:

```
do {
        wait(wrt);
        // waiting is performed
        signal (wrt);
} while (True)
```

The structure of a reader process:

```
do {
        wait(mutex);
        readcount++;
        if (readcount == 1)
                wait(wrt);
        signal mutex;
        // reading is performed
        wait(mutex);
        readcount--;
        if (readcount == 0)
                signal(wrt);
        signal(mutex);
} while (True)
```

Deadlocks

Deadlock Four Condition

Deadlock can arise if four conditions hold simultaneously.

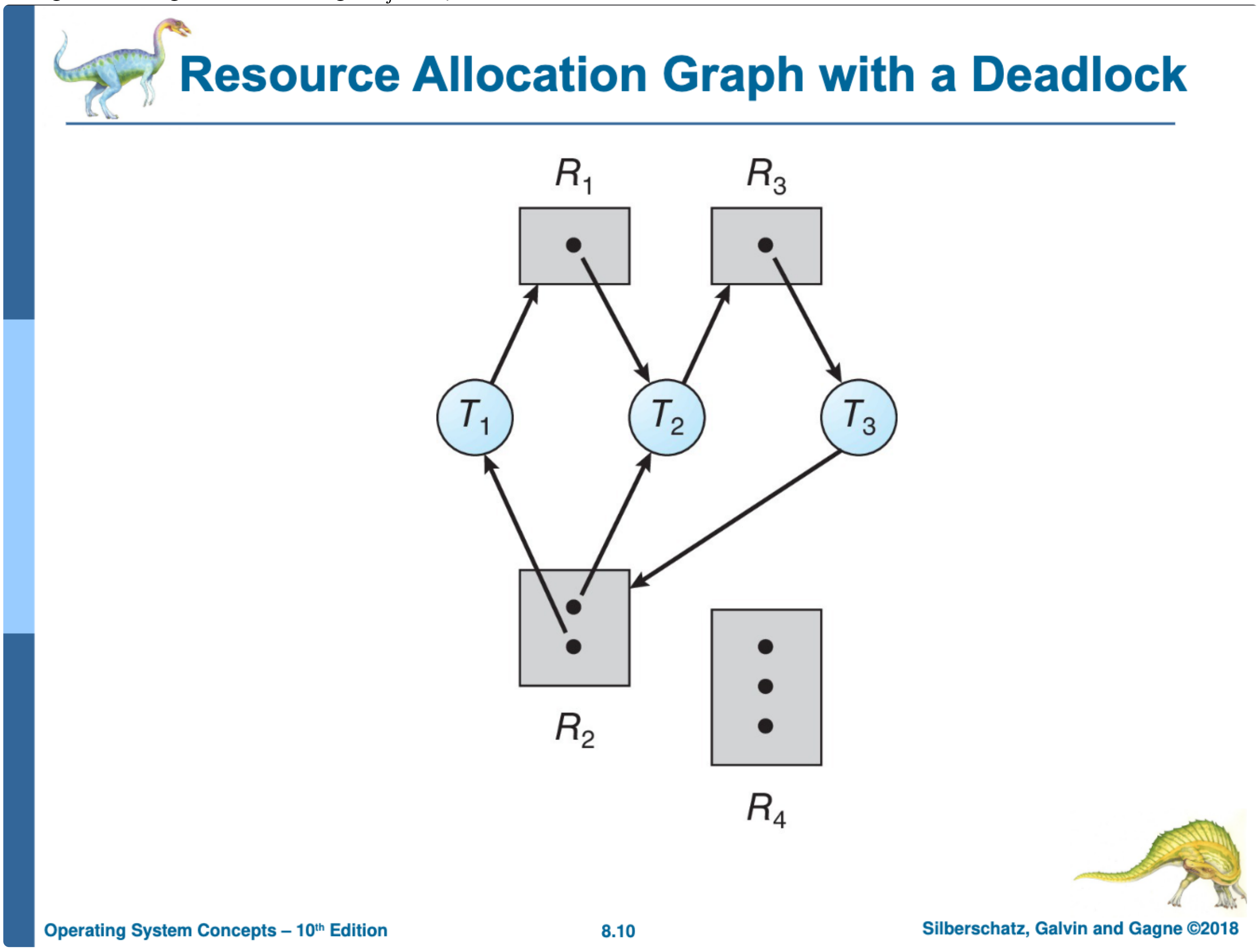
1. Mutual exclusion: only one thread at a time can use a resource.
2. Hold and wait: a thread holding at least one resource is waiting to acquire additional resources held by other threads.
3. No preemption: a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.
4. Circular wait: there exists a set $\{T_0, T_1, \ldots, T_n\}$ of waiting threads such that T_0 is waiting for a resource that is held by T_1, T_1 is waiting for a resource that is held by $T_2, \ldots, T_{n-1}$ is waiting for a resource that is held by T_n, and T_n is waiting for a resource that is held by T_0.

Resource Allocation Graph

A set of vertices V and a set of edges E, where V is partitioned into two types:

1. $T = \{T_1, T_2, \ldots, T_n\}$, i.e., the set consisting of all the threads in the system.
2. $R = \{R_1, R_2, \ldots, R_m\}$, i.e., the set consisting of all resource types in the system.
   and
3. Request edge: directed edge $T_i \to R_j$.

4. Assignment edge: directed edge $R_j \rightarrow T_i$.



Resource Allocation Graph with a Deadlock

If the graph is acyclic, then it has no deadlock.

If the graph is cyclic, then,

1. If only one instance per resource type, then deadlock.
2. If several instances per resource type, possibility of deadlock.

Deadlock Prevention

To invalidate one of the four necessary conditions for deadlock:

1. Mutual exclusion: not required for sharable resources (e.g., read-only files), and which must hold for non-sharable resources.
2. Hold and wait: must guarantee that whenever a thread requests a resource, it does not hold any other resources.
    1. Require threads to request and be allocated all its resources before it begins execution or allow thread to request resources only when the thread has none allocated to it.
    2. Low resource utilisation, but starvation is possible.
3. No preemption:
    1. If a process that is holding some resources requests another resource that cannot be immediately allocated to ti, then all resources currently being held are released.
    2. Preempted resources are added to the list of resources for which the thread is waiting.
    3. Thread will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
4. Circular wait:
    1. Impose a total ordering of all resource types, and require that each thread requests resources in an increasing order of enumeration.

2. Invalidating the circular wait condition is most common,
    1. Simply assign each resource a (i.e., mutex locks) a unique number, e.g., `mutex1 = 1`, `mutex2 = 2`.
    2. Resource must be acquired in order.

Deadlock Avoidance

Requires that the system has some additional a priori information available:

1. Simplest and the most useful model requires that each thread declare the maximum number of resources of each type that it may need.
2. The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
3. Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State

When a thread requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

System is in safe state if there exists a sequence $\langle T_1, T_2, \ldots, T_n \rangle$ of all the threads in the systems such that for each T_i, the resources that T_i can still request can be satisfied by currently available resources, and resources are held by all the T_j, with $j < 1$, i.e.,

1. If T_i resource needs are not immediately available, then T_i can wait until all T_j have finished.
2. When T_j is finished, T_i can obtain needed resources, execute, return allocated resources, and terminate.
3. When T_i terminates T_{i+1}, it can obtains its needed resources, and so on.

Therefore, if a system is in safe state, then there is no deadlocks, otherwise, there may has deadlock. Avoidance is to ensure that a system will never enter an unsafe state.

Banker's Algorithm

Suppose there is multiple instances of resources, each thread must has a priori claim maximum use. When a thread requests a resource, it may have to wait. When a thread gets all its resources, it must return them in a finite amount of time.

The following is safety algorithm,

1. Let `Work` and `Finish` be vectors of length m and n, respectively. Initialise `Work = Available` and `Finish[i] = false` for $i = 0,\ 1, \ldots, n-1$
2. Find an index i such that both
    1. `Finish[i] == false`
    2. `Need_i <= Work`
    3. If no such i exists, go to step 4
3. `Work = Work + Allocation_i`
    1. `Finish[i] = true`
    2. Go to step 2
4. If `Finish[i] == true` for all i, then the system is in a safe state.

The following steps is to find the resources,

1. Given `Allocation[i][j]` and `Max[i][j]`, to find `Need[i][j] = Max[i][j] - Allocation[i][j]`.

2. To find the `Available[i][j]`, given the first `Available[i][j]`, find the suitable `Need[i][j]`, then `Available[i][j] - Need[i][j]`, then `+ Max[i][j]` as the next `Available[i][j]`, and loop this until all threads is done.
3. The final `Availabe[i][j]` is for how many resources of type $A, B, C, D...$ are there.
4. The safe sequence is simply the order we use in step 2.

If there is a new request,

1. If `Request_i ⩽ Need_i`
   1. Go to step 2
   2. Otherwise, raise an error condition
2. If `Request_i⩽Available_i`
   1. Go to step 3
   2. Otherwise, T_i must wait
3. Have the system pretend to have allocated the requested resources to the thread T_i by modifying the state as follows:
   1. `Available = Available - Request_i`
   2. `Allocation_i = Allocation_i + Request_i`
   3. `Need_i = Need_i - Request_i`
   4. If safe, then the resources are allocated to T_i.
   5. Otherwise, T_i must wait.

Main Memory

Contiguous Allocation

Main memory is usually into two partitions:

1. Resident OS, is usually held in low memory with interrupt vector.
2. User process, is usually held in high memory
3. Each process contained in single contiguous section of memory

Relocation registers is used to protect user processes from each other, and from changing OS code and data.

1. Base register contains value of smallest physical address.
2. Limit register contains range of logical addresses, i.e., each logical address must be less than the limit register.
3. Memory management unit (MMU) maps logical address dynamically
4. It can allow actions such as kernel code being transient and kernel changing size.

Dynamic Allocation

Assume there is a request of size n from a list of free holes, there are some solution:

1. First-fit: allocate the first hole that is big enough.
2. Best-fit: allocate the smallest hole that is big enough, and it must search entire list, unless ordered by size.
   1. It produces the smallest leftover hole.
3. Worst-fit: allocate the largest hold, as it must also search entire list.
   1. It produces the largest leftover hole.

Fragmentation

External fragmentation: the total memory space exists to satisfy a request, but it is not contiguous.

Internal fragmentation: the allocated memory may be slightly larger than the request memory, and the size difference is memory internal to a partition, but not being used.

First-fit analysis reveals that given N blocks allocated, $0.5N$ blocks lost to fragmentation, as 50 percent rule.

To reduce external fragmentation by compaction:

1. Shuffle memory contents to place all free memory together in one large block.
2. Compaction is possible only if relocation is dynamic, and it is done at execution time.
3. I/O problem:
   1. Latch job in memory while it is involved in I/O.
   2. Do I/O only into OS buffers.

Paging

The physical address space of a process can be noncontiguous, i.e., process is allocated physical memory whenever the latter is available:

1. Avoids external fragmentation.
2. Avoids problem of varying sized memory chunks.

It divides physical memory into fixed-sized blocks called frames, and its size if power of 2, between 512 bytes and 16 Mbytes.

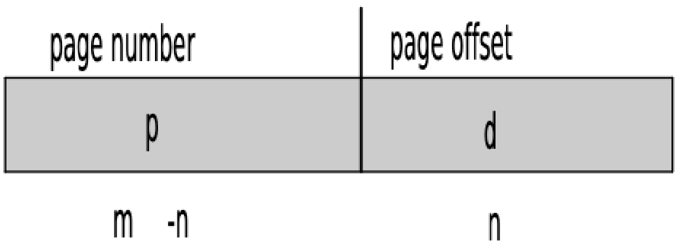
It divide logical memory into blocks of same size called pages.

To run a program of size N pages, then it is needed to find N free frames and load program.

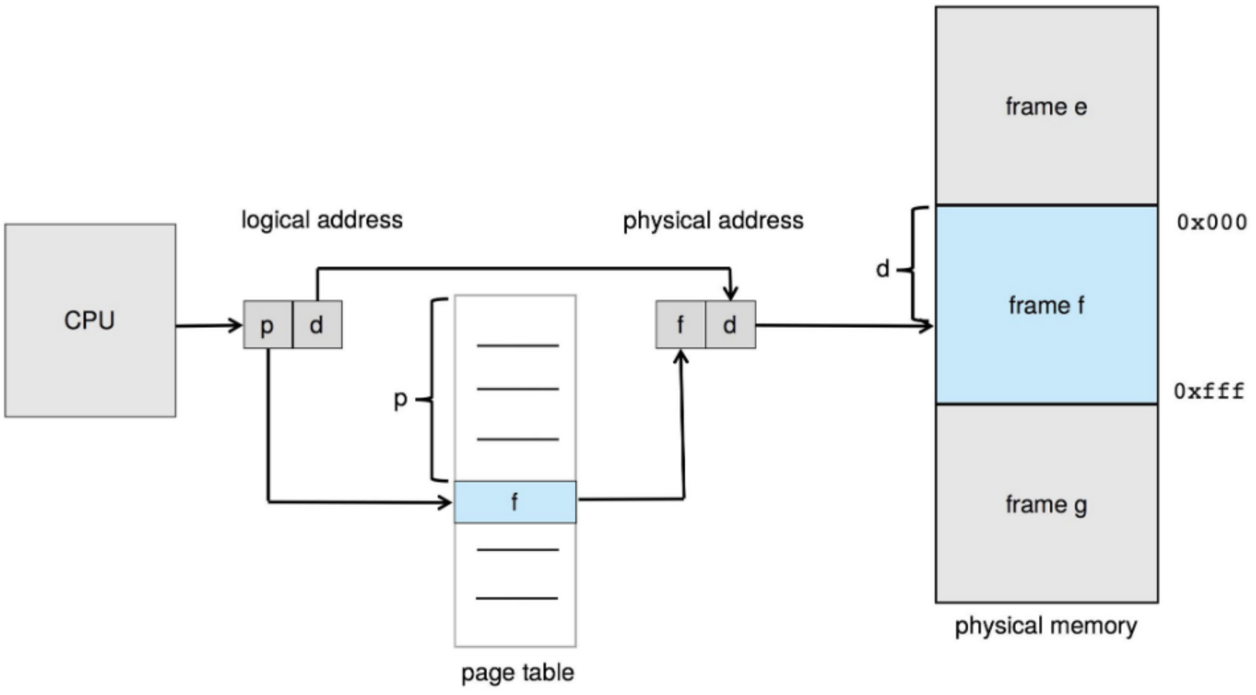
To set up a page table to translate logical to physical addresses.

Page number p is used as an index into a page table which contains base address of each page in physical memory.

Page offset d is combined with base address to define the physical memory address that is sent to the memory unit.



- For given logical address space 2^m and page size 2^n



The following steps is to calculate the internal fragmentation:

1. Given page size x and process size y.
2. Then the number of pages is $\lfloor y/x \rfloor$ + remaining bytes.
3. Internal fragmentation $= x - \text{remaining bytes}$.
4. Worst case fragmentation $= 1 \text{ frame} - 1 \text{ byte}$.
5. Average fragmentation $= 1/2 \text{ frame size}$.

The following steps is to find the the memory of physical address space for the page table alone:

1. Given a 32 bits logical address space as on modern computers and page size of 4KB (2^{12}).
2. The size of the page table would have $2^{32}/2^{12}$.
3. If each entry is 4 bytes, then each process would have $4 \cdot 2^{32-20}$ bytes of physical address space for the page table alone.

The following steps is to translate the virtual addresses to physical addresses, e.g.,

1. Given address space size 16k, physical memory size 64k, page size 4k,
2. The format of the page-table entry is $\text{valid bit(left-most)}|\text{PFN}$,
3. For example, $\text{VA } 0\text{x}3229 \rightarrow 0011\ 0010\ 0010\ 1001$.

4. We know that there are $\log_2(16 \cdot 1024) \to 14 \text{ bits}$ in the virtual address, and that the offset $\log_2(4 \cdot 1024) = 12 \text{ bits}$, then leaving $14 - 12 = 2 \text{ bits}$ for the VPN.
5. From $0011\ 0010\ 0010\ 1001$, $0011 \to \text{virtual page } 3$ with an $\text{offset } 0\text{x}229$
6. Now, look at the page table 3: $0\text{x}8000006 \to 1 \ldots 0010$, which is valid from step 2, with $\text{physical page } 6$
7. Then, map $\text{VA } 0\text{x}3229 \to \text{physical page } 6$.
8. Finally, $\text{physical page} + \text{offset} \to 0\text{x}6229$.

Speed up Accessing Page Table (TLBs)

Page table is kept in main memory,

1. Page-table base register (PTBR) points to the page table.
2. Page-table length register (PTLR) indicates the size of the page table.
3. Translation look-aside buffers (TLBs) store address-space identifiers (ASIDs) in each TLB entry: it uniquely identifies each process to provide address space protection for that process.
    1. Otherwise, it needs to flush at every context switch
    2. It is typically small (64 to 1024 entries)
    3. On a TLB miss, value is loaded into the TLB for faster access next time.
        1. Replacement policies must be considered, some entries can be wired down for permanent fast access.

Effective Access time (EAT)

Hit ratio is a percentage of times that a page number is found in the TLB.

1. It it takes x nanoseconds to access memory
    1. A mapped-memory access takes x nanoseconds to access memory
        1. When the page number is in the TLB
    2. If we fail to find the page number in the TLB
        1. We must first access memory for the page table and frame number, which takes x nanoseconds
            1. Then access the desired byte in memory, which also takes x nanoseconds
        2. Therefore, it takes $2x$ nanoseconds

$$\text{EAT} = \text{hit ratio} \times x + (1 - \text{hit ratio}) \times 2x \tag{1}$$

Multilevel Paging

Advantage:

1. Only allocates page-table space in proportion to the amount of address space you are using.
2. The OS can grab the next free page when it needs to allocate or grow a page table.

The following steps is to translate $\text{VA} \to \text{PA}$ with a multi-level page table:

1. Given page size 32 bytes, address space 32 KB, physical memory consists of 128 pages.
2. VA has $\log_2(32 \cdot 1024) = 15$ bits.
3. offset has $\log_2(32) = 5$ bits. (32 is the page size)
4. PA has $\log 2(128 \cdot 32) = 12$ bits, with $12 - 5 = 7$ bits for PFN and 5 bits for offset.
5. $\text{VPN} = 15 - 5 = 10$ bits, therefore 5 bits for PDE and 5 bits for PTE
6. As the page directory needs one entry per page of the page table, it has 32 page-directory entries (PDEs).
7. $\text{PDE} : \langle \text{valid} | PT_6 \ldots PT_0 \rangle$, as if it is invalid (0), then raise an exception.
8. Given a the value of the page directory base register (PDBR), which tells you which page is the page directory is located upon.
9. First, to locate the page number which is same to PDBR.

10. Suppose, to translate $\text{VA } 0\text{x}611\text{c} \to 11000\ 01000\ 11100$.
11. $\text{PDE} \to 11000 \to 0\text{x}18 \to 24_{10}$, now look at the page PDBR 24-th entry, giving content $a1 \to 1\ 0100001$ with valid bit 1, then $\text{PFN } 0100001 \to 0\text{x}21 \to 33_{10}$
12. Now locate to page 33, with $\text{PTE} = 01000 \to 0\text{x}8 \to 8$, look at the page 33 8-th entry, giving content $b5 \to 1\ 0110101$ with valid bit 1, then $\text{PFN } 0110101 \to 0\text{x}35 \to 53$.
13. Now combine $0110101\ 11100 \to 0\text{x}6\text{bc}$, now locate the page 53 28-th entry, giving $08 \to 00001000 \to 8_{10}$.

Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution, as total physical memory space of processes can exceed physical memory.

Backing store: a fast disk large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.

Roll out, roll in: a swapping variant used for priority-based scheduling algorithms, as lower-priority process is swapped out so higher-priority process can be loaded and executed.

Major part of swap time is transfer time, total transfer time is directly proportional to the amount of memory swapped.

System maintains a ready queue of ready-to-run processes which have memory images on disk.

Segmentation

It is a memory-management scheme that supports user view of memory, logical address consists of a two tuple: $\langle\text{segment-number, offset}\rangle$.

Segment table: it maps two-dimensional physical addresses, each table entry has:

1. Base: it contains the starting physical addresses where the segments reside in the memory.
2. Limit: it specifies the length of the segment.

Segment-table base register (STBR) points to the segment table's location in memory.

Segment-table length register (STLR) indicates the number of segments used by a program. As segment number s is legal if $s < \text{STLR}$.

Virtual Memory

Demand Paging

Demand paging is a paging system with swapping. When we want to execute a program on secondary storage (disk), we swap it into memory. Instead of swapping in entire program into physical memory we only swaps those pages next to the executing instruction.

A page table is used to keep track of pages that are in physical memory (valid 1) or on disk (valid 0). If the page is not currently in physical memory then the address in back storage is also stored in page table.

If the program tried to use a page that is not in memory, a page fault trap will occur (caused by the invalid bit in page table).

Demand paging bring a page into memory only when it is needed:

1. Less I/O needed.

2. Less memory needed.
3. Faster response.
4. More users.

Given memory access time a, and average page-fault service time b

$$\text{EAT} = (1-p)a + bp \tag{2}$$

If we want the performance degradation is less than 10 percent,

$$0.1a + a > EAT \tag{3}$$

Then, find the optimal value of p

To Handle Page Fault

1. If there is a reference to a page first reference to that page will trap to OS.
   1. Page fault.
2. OS looks at another table to decide:
   1. Invalid reference ⇒ abort.
   2. Just not in memory.
3. Find free frame.
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory.
   1. Set validation bit ⇒ `v`.
6. Restart the instruction that caused the page fault.

The Stages of Demand Paging in Worst case

1. Trap to the OS.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page references was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
   1. Wait in a queue for this device until the read request is serviced.
   2. Wait for the device seek and/or latency time.
   3. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user.
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupt instruction.

Page Replacement

To prevent over-allocation of memory by modifying page-fault service routine to include page replacement.

To use modify (dirty) bit to reduce overhead of page transfers: only modified pages are written to disk.

Page replacement completes separation between logical memory and physical memory, as large virtual memory can be provided on a smaller physical memory.

1. Find the location of the desired page on disk
2. Find a free frame:
    1. If there is a free frame, use it.
    2. If there is no free frame, use a page replacement algorithm to select a victim frame.
        1. Write the victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame, and update the page and frame tables.
4. Continue the process by restarting the instruction that caused the trap.

There are several paging replacement algorithm:

1. FIFO
2. Optimal Algorithm
    1. Replace the page that will not be used for longest period of time (future predicting)
3. Least recently used (LRU)
    1. Replace the page that has not been used in the most amount of time
    2. Counter implementation:
        1. Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter.
        2. When a page needs to be changed, look at the counters to find smallest value.
            1. Search through table needed.
    3. Stack implementation:
        1. Keep a stack of page numbers in a double link form:
        2. Page referenced:
            1. Move it to the top.
            2. Requires 6 pointers to be changed.
        3. But each update is more expensive.
        4. No search for replacement.
4. Lease frequently used (LFU) with Counting Algorithm.
    1. Replaces page with smallest count.
5. Most frequently used (MFU).

Thrashing

If a process does not have enough pages, the page-fault rate is very high:

1. Page fault to get page.
2. Replace existing frame, but quickly need replaced frame back.
3. This leads to:
    1. Low CPU utilisation.
    2. OS thinking that it needs to increase the degree of multiprogramming.
    3. Another process added to the system

Thrashing that a process is busy swapping pages in and out.

Page Size

Page size selection must take into consideration:

1. Fragmentation: large page size increases internal fragmentation
2. Page table size: large page size reduces page table size
3. Resolution
4. I/O overhead: large page size increases the amount of I/O
5. Number of page faults: large page size reduce the number of page faults

6. Locality
7. TLB size and effectiveness

It is always power of 2, and average, it grows over time.

File System Interface

FS Definition

It is collection of files and a directory structure. In the directory structure, it is a collection of nodes containing information about all files and both directory structure and the files reside on disk. It provides user interface to storage, mapping logical to physical, and provides efficient and convenient access to disk by allowing data to be stored, located retrieved easily.

Directories Structure

To guarantee there is no cycle:

1. Allow only links to files not subdirectories/
2. Garbage collection: run a periodical process to clean up after a period of time.
3. Every time a new link is added use a cycle detection algorithm to determine whether it is ok.

Access lists and groups in Unix: $761 \to 111\ 110\ 001 \leftrightarrow \text{rwx}$.

Ch11.

File Control Block (inode)

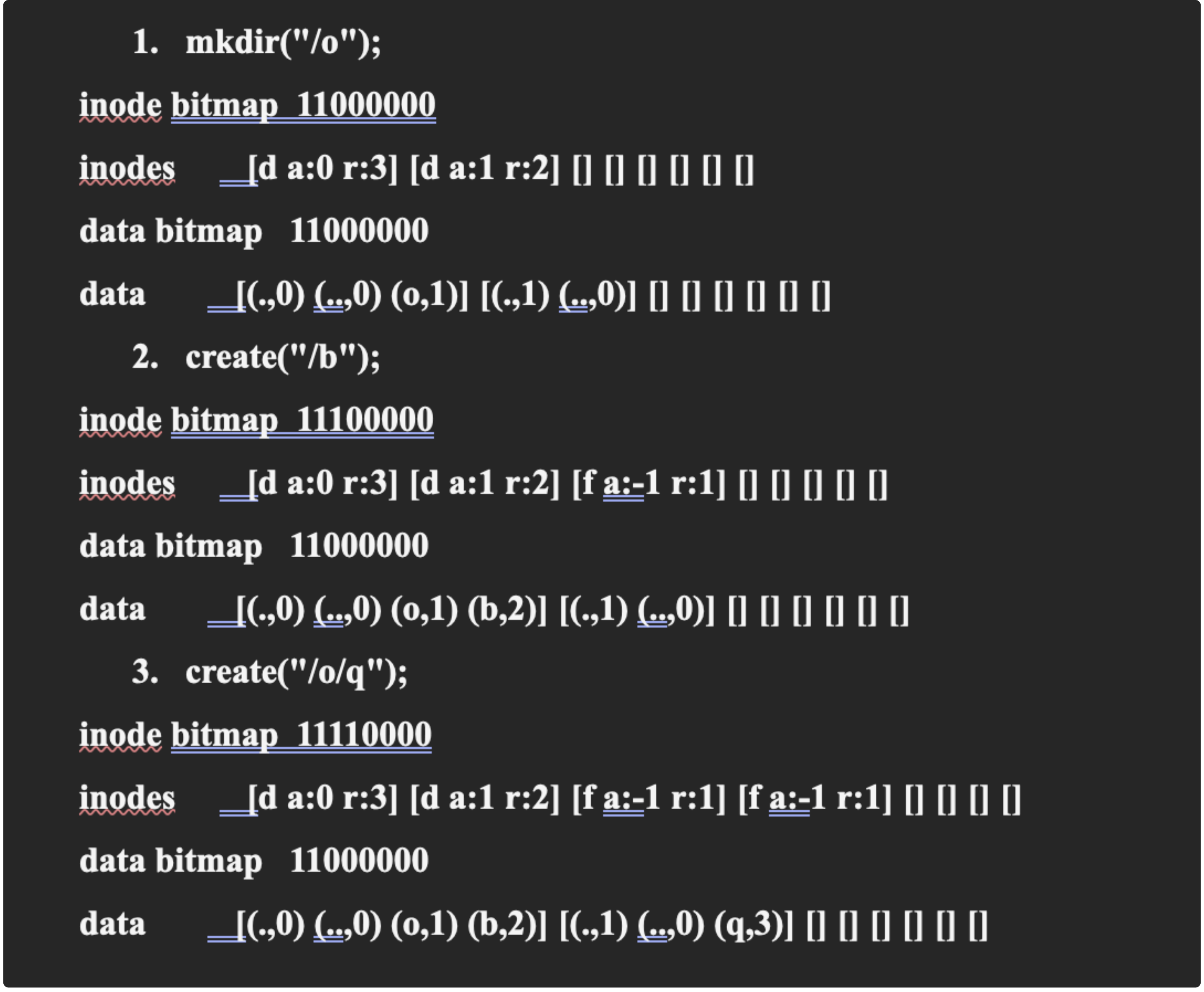
It is a storage structure consisting of information about a file.

1. Inode bitmap: indicates which inodes are allocated
   1. 1 indicating it is allocated
   2. 0 indicating it is free
2. Inode: tables of inodes and their contents with three fields
   1. The type of file: f for a regular file, d for a directory
   2. Which data bock belongs to a file: -1 for empty, non-negative address for one block in size
   3. Reference count for the file or directory
3. Data bitmap: indicates which data blocks are allocated
4. Data: indicates contents of data blocks with two fields
   1. (name, inumber)
   2. If it contains user data: $[x]$

$$[\text{f a:-1 r:1}] \tag{4}$$

where -1 means it is empty, and now with block 10 allocated:

$$[\text{f a:10 r:1}] \tag{5}$$



```
1. mkdir("/o");
inode bitmap 11000000
inodes      [d a:0 r:3] [d a:1 r:2] [] [] [] [] [] []
data bitmap  11000000
data        [(.,0) (..,0) (o,1)] [(.,1) (..,0)] [] [] [] [] [] []
2. create("/b");
inode bitmap 11100000
inodes      [d a:0 r:3] [d a:1 r:2] [f a:-1 r:1] [] [] [] [] []
data bitmap  11000000
data        [(.,0) (..,0) (o,1) (b,2)] [(.,1) (..,0)] [] [] [] [] [] []
3. create("/o/q");
inode bitmap 11110000
inodes      [d a:0 r:3] [d a:1 r:2] [f a:-1 r:1] [f a:-1 r:1] [] [] [] []
data bitmap  11000000
data        [(.,0) (..,0) (o,1) (b,2)] [(.,1) (..,0) (q,3)] [] [] [] [] [] []
```

Allocation Methods

An allocation methods refers to how disk blocks are allocated for files.

There are three major methods:

1. Contiguous Allocation Method: each files occupies set of contiguous blocks.
   1. Best performance in most cases.
   2. Simple: only starting location (block number) and length (number of blocks) are required.
   3. Mapping from logical to physical, e.g., block size 512 bytes, block to be accessed equal to the starting address $+Q$, displacement into block is R
2. Linked allocation: each file is a linked list of blocks.
   1. File ends at `nil` pointer.
   2. No external fragmentation.
   3. Each block contains pointer to next block.
   4. No compaction, external fragmentation.
   5. Free space management system called when new block needed.
   6. Improve efficiency by clustering blocks into groups but increases internal fragmentation.
   7. Reliability can be a problem.
   8. Locating a block can take many I/Os and disk seeks.
   9. Mapping: block to be accessed is the Q^{th} block in the linked chain of blocks representing the file, displacement into block is $R+1$.
3. Indexed allocation: brings all pointers together into the index block.
   1. It needs index table.

2. Random access.
3. Dynamic access without external fragmentation, but have overhead of index block.

Mass Storage Systems

Hard Disk Drivers Structure

HDDs spin platters of magnetically-coated material under moving read-write heads:

1. Drives rotate at 60 to 250 times per second.
2. Transfer rate is rate at which data flow between drive and computer.
3. Positioning time (random-access time) is time to move disk arm to desired cylinder (seek time) and time for desired sector to rotate under the disk head (rotational latency).
4. Head crash results from disk head making contact with the disk surface

Hard Disk Performance

Access latency = average access time = average seek time + average latency.

$$\text{Access time} = \text{Seek time} + \text{Rotational latency} + \text{Transfer time} \tag{6}$$

HDD Scheduling

To minimise the seek time as well as the 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 the last transfer.

E.g., a request queue (0-199): $98, 183, 37, 122, 14, 124, 65, 67$ with a head pointer 53,

1. FCFS
2. Shortest seek time first (SSTF)
3. SCAN (Elevator): starts at one end of the disk, and moves toward the other end.
4. C-SCAN: when it reaches the other end, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.
5. C-LOOK: it does not go to the end, but the last request in each direction.

HDD Calculation

Assume that each track contains 12 sectors. The disk head is currently positioned over sector 6 of the outermost track. The direction of rotation is counter-clockwise, and it takes 360 time units.

FCFS:

1. 360/12 = 30 per sector
2. Block 3 → seek: 0 rotate: 8.5×30 transfer: 30 total: 285
3. Block 14 → seek: 40 rotate: 10×30-40 transfer: 30 total: 330
4. Block 4 → seek: 40 rotate: 360+(1×30)-40 transfer: 30 total: 420
5. Block 17 → seek: 40 rotate: 12×30-40 transfer: 30 total: 390

SSTF:

1. Block 3 → seek: 0 rotate: 8.5×30 transfer: 30 total: 285

2. Block 4 → seek: 0 rotate: 0 transfer: 30 total: 30
3. Block 14 → seek: 40 rotate: 9×30-40 transfer: 30 total: 300
4. Block 17 → seek: 0 rotate: 2×30 transfer: 30 total: 90