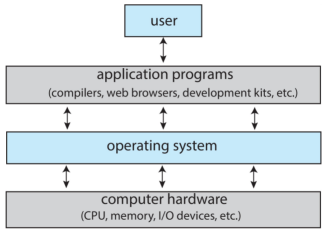
OS Course Summary

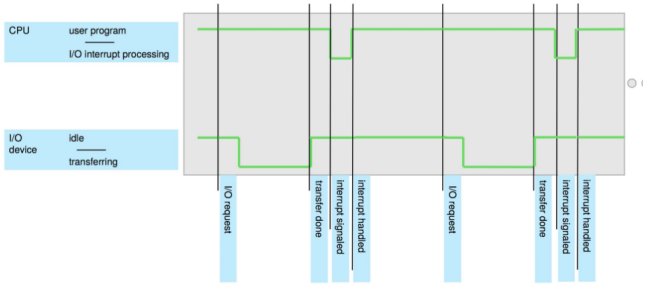
Summary done by Voiades Robert and Robu Petru.

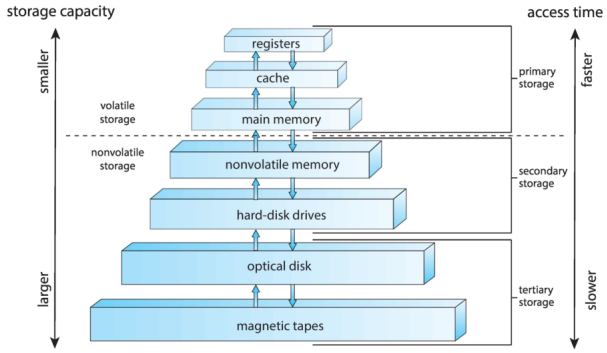
1 Operating System

OS = coordiantes use of hardware among various applications and users.



I/O devices and CPU can execute concurrently. The devices communicate with the CPU through **interrupts**.





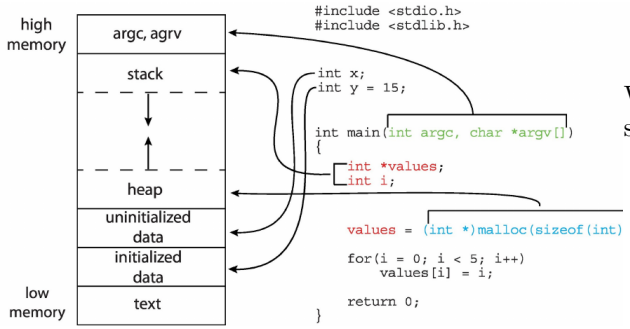
2 Processes

2.1 General

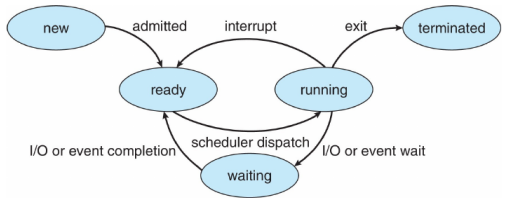
Process = program (executable / binary) in execution.

Program = passive entity stored on disk; Process = active entity loaded in memory.
Parts of a process:

- Text section (program code)
- Current activity (program counter and process registers)
- Stack (function parameters, local vars, ret addresses)
- Data section (global variables)
- Heap (dynamically alocated memory)



Process states: New, Running, Waiting, Ready, Terminated.

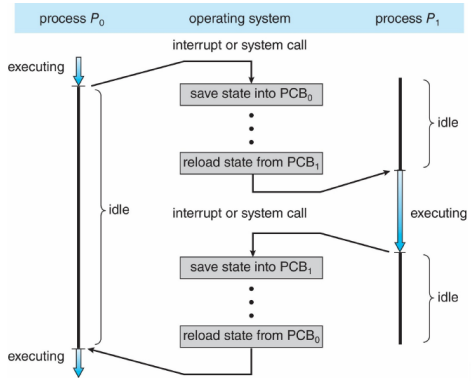


2.2 PCB

Information associated with each process (also called task control block)
PCB contents:

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

When CPU switches context, the respective state stored in the PCB is loaded:



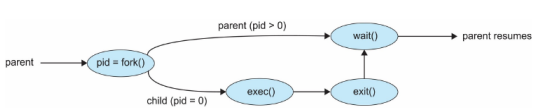
2.3 Process creation / termination

Parent process create children processes, which, in turn create other processes, forming a tree of processes.
Process is identified and managed via a process identifier (pid).

- fork() = syscall to create a new child process. Child's adress space is the same as parent. Returns 0 in the child process and PID of the child in parent.

- exec() = replace the child's adress space with another program.
- wait() = parent process calls wait() to wait for the child to terminate.
- exit() = process executes last statement and then asks the operating system to delete it.
- abort() = terminate the execution of children processes.
- waitpid() = waits for a specific child spawned by the process.

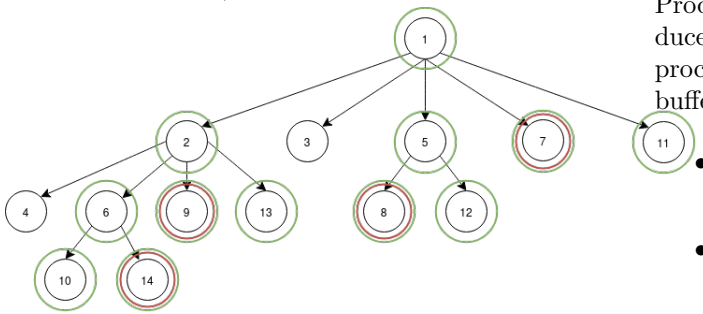
```
int main(int argc, char *argv[])
{
    pid_t pid = fork();
    if(pid < 0)
        return errno;
    else if(pid == 0)
    {
        //child
        char *argv[] = {"ls", NULL};
        execve("/bin/ls", argv, NULL);
        perror(NULL);
    }
    else
    {
        //parent
        wait(NULL);
        // child finished
    }
    return 0;
}
```



Exemplu arbore de procese: Câte procese și thread-uri sunt la final? Desenați arborescența de procese și thread-urile aferente.

```
fork()
if (fork()) {
    fork()
    if (!fork())
        pthread_create()
    else
        fork()
    pthread_create()
}
```

Sunt 14 procese si 16 thread-uri. Fiecare cerc este un thread format (prima data cele rosii, la linia 5, apoi cele verzi la linia 8).

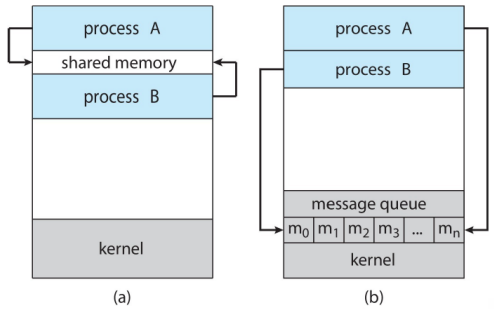


Zombie and Orphan processes:

- Zombie process = a process that has completed execution but still has an entry in the system process table. This happens because the parent didn't invoke wait().
- Orphan process = a living child whose parent has died. Could be because parent exited prematurely or intended in the case of daemons. The process is adopted by process with PID=1.

2.4 Interprocess Communication

Processes within a system may be independent or cooperating. Cooperating processes can affect or be affected by other processes for: sharing data, computation speedup, modularity, convenience.
Two models of IPC: Shared memory and Message passing.



2.4.1 Shared memory

Requires careful synchronization.
Producer-Consumer problem: producer process produces information that is consumed by a consumer process. Solved with shared memory by holding a buffer and in/out pointers.

- unbounded-buffer = Producer never waits; Consumer waits if there is no buffer to consume
- bounded-buffer = Producer must wait if all buffers are full; Consumer waits if there is no buffer to consume

Shared memory used in UNIX with shm_open() to create a shared memory segment and mapped it to a file descriptor via mmap.

2.4.2 Message passing

Processes communicate with each other without resorting to shared variables.
IPC facility provides two operations: **send(message)** and **receive(message)**

- Direct Communication = Processes must name each other explicitly: send (P, message) - send a message to process P, receive(Q, message) - receive a message from process Q
- Indirect Communication = Send and receive messages through mailbox: send(A, message) - send a message to mailbox A, receive(A, message) - receive a message from mailbox A

Message passing may be either blocking or non-blocking:

- Blocking (synchronous)
  - Blocking send = the sender is blocked until the message is received
  - Blocking receive = the receiver is blocked until a message is available
- Non-blocking (asynchronous)

- Non-blocking send = the sender sends the message and continue
- Non-blocking receive = the receiver receives a valid message / NULL

If both send and receive are blocking, we have a **rendezvous**.
As processes can not be synchronized perfectly, we make use of buffering (queue of messages attached to the link):

- Zero capacity: no messages are queued on a link. Sender must wait for receiver. (rendezvous)
- Bounded capacity: finite length of n messages. Sender must wait if link full.
- Unbounded capacity: infinite length. Sender never waits.

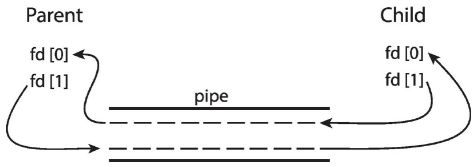
2.4.3 Signals

Signals can be send to a process:

- SIGINT (ctrl+c) = kills the process
- SIGSTOP (ctrl+z) = stops the process, moves it to background
- SIGSEGV = invalid memory accesed
- etc.

2.4.4 Pipes

Acts as a conduit allowing two processes to communicate.
Ordinary pipes (anonymous) = cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
Named pipes = can be accessed without a parent-child relationship.



2.4.5 Sockets

A socket is defined as an endpoint for communication. Concatenation of **IP address and port** (a number included at start of message packet to differentiate network services on a host).
loopback (127.0.0.1) = system on which process is running

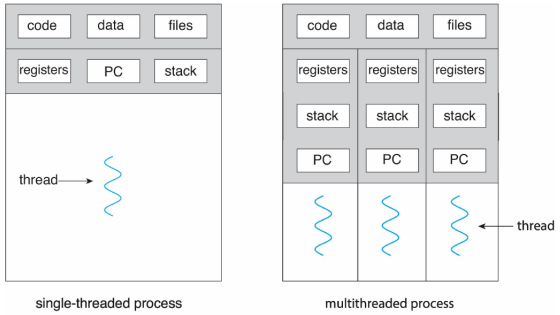
2.4.6 RPCs

RPCs = remote procedure calls. Abstracts procedure calls between processes on networked systems Stubs = client-side proxy for the actual procedure on the server The client-side stub locates the server and marshalls the parameters. The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

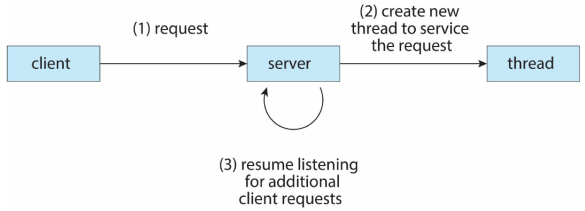
3 Threads

General

A thread is the basic unit of CPU utilization. It is less costly than process, because each process can have multiple threads that share resources (data, code, files), having only independent registers, stack and PC. (the multithreaded model).



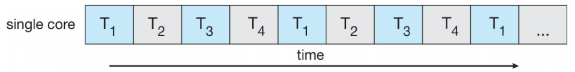
They can be used in the server model to process requests



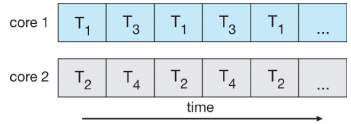
Parallelism & concurrency

- **Parallelism** = a system can execute more than one task simultaneously.
- **Concurrency** = more than one task is making progress.

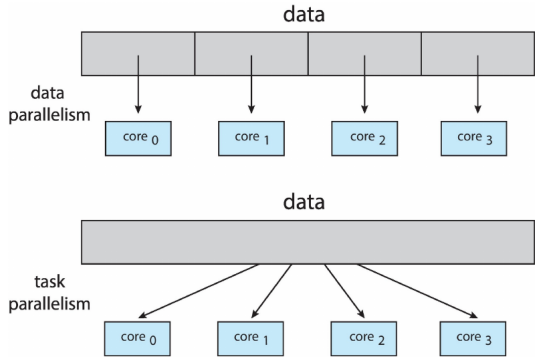
- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



- **data parallelism** = distribute subsets of same data across multiple cores, same operation
- **task parallelism** = distribute threads across multiple cores, each with unique operation



Amdahl's law

This measures the performance gain from adding cores to an application that has both sequential and parallel components.
- S - serial portion (fraction), $1 - S$ - parallel portion
- N - cores

$$speedup \leq \frac{1}{S + \frac{1-S}{N}}$$

Applied. $S = 0.25$ (25% serial), moving from 1 to 2 cores $N = 2$.

$$s \leq \frac{1}{0.25 + \frac{0.75}{2}} = 1.6$$

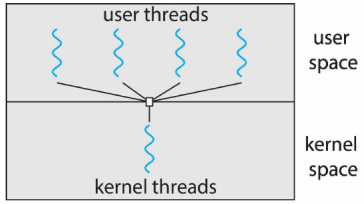
Speedup of up to 1.6 times.

User vs Kernel Threads

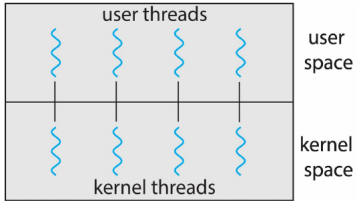
- **User thread** = managed by user level library (e.g. pthread)
- **Kernel Thread** = supported by the Kernel

There are three models that map user threads to kernel threads.

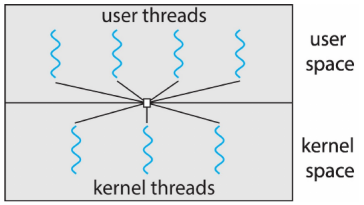
- **many-to-one**: many user threads mapped to 1 kernel thread. One thread blocking causes others to block.



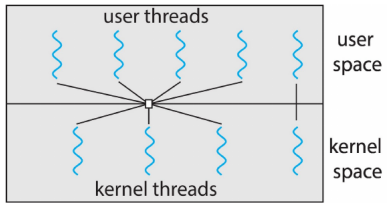
- **one-to-one**: 1 user threads mapped to 1 kernel thread. More concurrency than previous.



- **many-to-many**: many user threads mapped to many kernel threads. Allows OS to create sufficient number of kernel threads.



- **two level**: similar to M:M but allows one user thread to be bound to kernel thread.



Other

Implicit threading represents the growing trend of having threads created and managed by compilers and run-time libraries, instead of creating them explicitly. Many methods.

- thread pools : create a pool of threads that await work
- fork-join parallelism: like divide et impera, but on threads

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

- OpenMP - set of compiler directives that identify parallel regions `#pragma omp parallel`
- there exists a complication on **fork** and **exec**. - there exists some versions of fork (one that forks all threads, one that forks only the thread that calls the fork) - exec is simpler and it replaces all the threads with the new code.

Signal

Signals are used in UNIX systems to notify a process that something has happened.

A signal handler is used to process a signal. We have predefined and user defined handlers.

A signal can be delivered to all threads, to certain threads or the thread that the signal applies to.

- **thread cancellations** is the termination of a thread before it finishes. There are two approaches **asynchronous cancellation** - terminate immediately and **deffered cancellation** allows the thread to check periodcally if it should be cancelled.

Cancellation can be disabled by the thread.

- **Thread Local Storage** - each thread can have its own copy of data

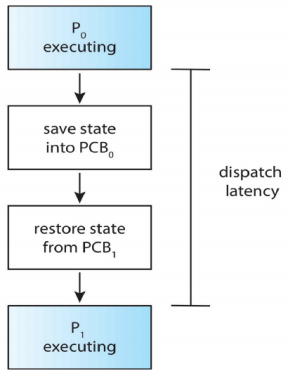
4 CPU Scheduling

How to schedule processes on the CPU efficently? Usually, CPU is a sequence of CPU bursts and I/O bursts (waiting time for I/O). A CPU scheduler selects a process and allocates a core to it.

- **Preemptive** = can postpone processes (put process in ready state / back in running state)
- **Non-preemptive** = once process has CPU, the process keeps it until the end or by switching to waiting.

Dispatcher = gives control of the CPU to the process selected by the CPU scheduler: switching context, switching to user mode, jumping to proper location in the user program to restart it.

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



Scheduling criteria:

- CPU utilization – keep the 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
- 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.

Note: Waiting Time $= t_f - t_s -$ burst.

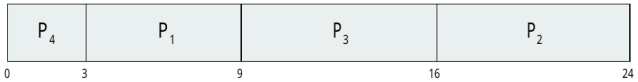
4.1 FCFS

FCFS = First-Come, First-Served Put the processes on the CPU the order in which they request it. (this is non-preemptive) Convoy effect - short process behind long process (this results in optimal average waiting time)
Example: For processes with burst times $P_1 = 24$; $P_2 = 3$; $P_3 = 3$; This is the resulting Gantt Chart:

If the processes come in different order, it is better.

4.2 SJF

SJF = Shortest Job First Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time. Optimal average waiting time.
Non-Preemptive version: For processes with burst times $P_1 = 6$; $P_2 = 8$; $P_3 = 7$; $P_4 = 3$ This is the resulting Gantt Chart:

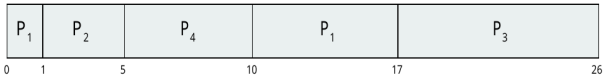


Determining the length of the next CPU-burst: ask the user to provide estimation / estimate ourselves.
Exponential averaging = Use length of previous CPU-burst and estimate the next one:

t_n = length on n-th CPU burst
τ_{n+1} = predicted next CPU burst
$\alpha, 0 \geq \alpha \leq 1$. Commonly $\alpha = \frac{1}{2}$
$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

4.3 SRT

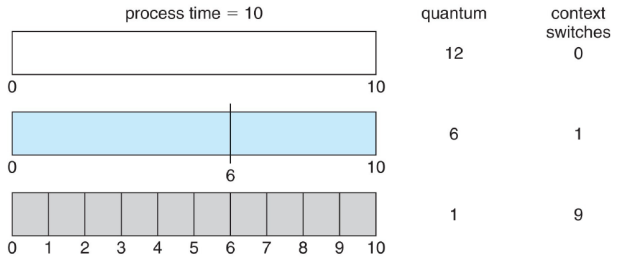
SRT = Shortest remaining time first
Preemptive version of SJF: For processes with arrival, burst times $P_1 = 0 \quad 8$; $P_2 = 1 \quad 4$; $P_3 = 2 \quad 9$; $P_4 = 3 \quad 5$ This is the resulting Gantt Chart:



4.4 Round-Robin (RR)

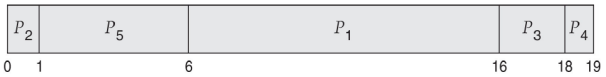
Each process gets a small unit of CPU time (time quantum q). After this time has elapsed, the process is preempted and added to the end of the ready queue.
Preemptive RR: For processes with burst times: $P_1 = 24$; $P_2 = 3$; $P_3 = 3$ and q = 4, the Gantt chart is:

Choose q carefully, take into account context switches, and waiting time of processes:



4.5 Priority Scheduling

A priority number (integer) is associated with each process. The CPU is allocated to the process with the highest priority (smallest integer = highest priority).
SJF is priority scheduling.
Starvation = low priority processes may never execute. Can fix with **aging** = as time progresses increase the priority of the process.
Non Preemptive Priority Scheduling: For processes with burst times and priority: $P_1 = 10 \quad 3$; $P_2 = 1 \quad 1$; $P_3 = 2 \quad 4$; $P_4 = 1 \quad 5$; $P_5 = 5 \quad 2$, the Gantt chart is:

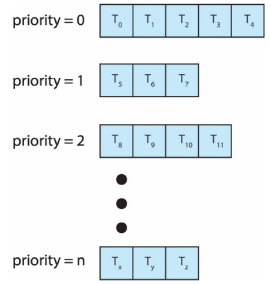


4.6 Priority scheduling with Round Robin

Run the process with the highest priority. Processes with the same priority run round-robin.
Example: For processes with burst and priority: $P_1 = 4 \quad 3$; $P_2 = 5 \quad 2$; $P_3 = 8 \quad 2$; $P_4 = 7 \quad 1$; $P_5 = 3 \quad 3$ and $q = 2$

4.7 Multilevel Queue

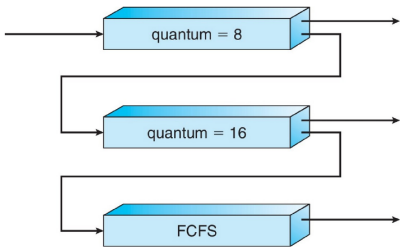
The ready queue consists of multiple queues.
Each queue has it's own scheduling algorithm + method to determine which queue a process goes to.
Example, multilevel queue by priority:



Multilevel Feedback Queue = A process can move between the various queues + method to determine

which queue a process goes to + method to determine when to upgrade/demote process.
Example (Three queues):

- Q_0 = RR with q = 8ms
- Q_1 = RR with q = 16ms
- Q_2 = FCFS



When a process gains CPU, the process receives 8 ms. If it does not finish in 8 milliseconds, the process is moved to Q_1. If the process doesn't finish in 16 ms, it goes to FCFS.

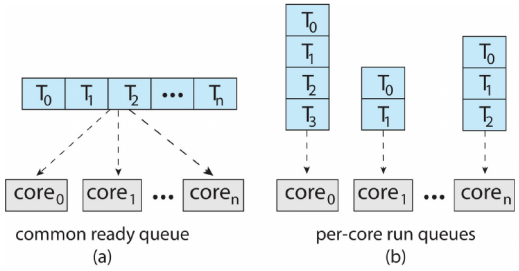
4.8 Thread scheduling

When threads are supported, threads are scheduled, not processes. Use the same algorithms.
Process-contention scope (PCS) = competition within the process. (on many-to-many / one-to-many threading models)
System-contention scope (SCS) = competition among all threads in system (on one-to-one models)

4.9 Scheduling on paralell systems

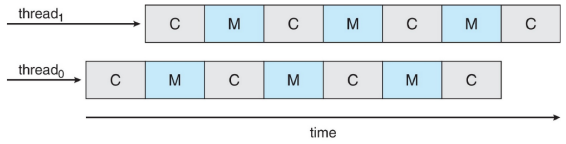
4.9.1 Multiple-Processor Scheduling

Symmetric multiprocessing (SMP) = each processor is self scheduling.
All threads may be in a common ready queue (a) or each processor may have its own private queue of threads (b).



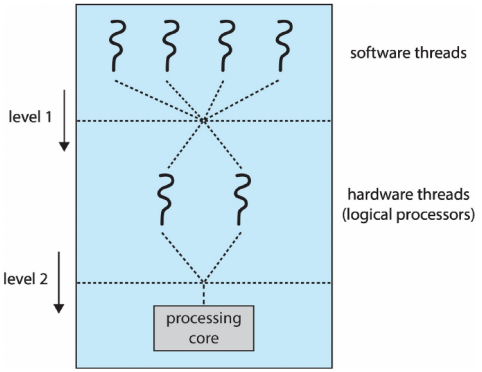
4.9.2 Multithreaded Mulitcore systems

Each core has 1 hardware threads. Then if one thread stalls, switch to another one. If they end up interleaved we have a speedup.



This is called Chip-multithread (hyperthreading) when you have multiple hardware threads on multiple cores. The systems sees each hw thread as a logical processor.
There are two levels of scheduling here:

- 1. The operating system deciding which software thread to run on a logical CPU
- 2. How each core decides which hardware thread to run on the physical core.



4.9.3 Load Balancing

: If the arhitecture is SMP, we need to keep all CPUs loaded for efficiency:

- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

4.9.4 Processor Affinity

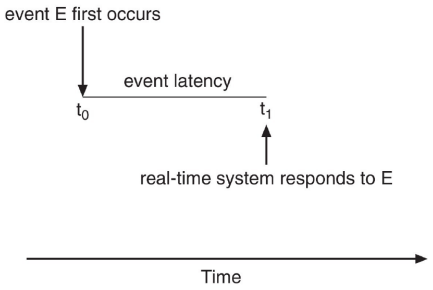
When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread. This is called **Affinity**.
Load balancing may affect affinity.
Solutions:

- Soft affinity – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- Hard affinity – allows a process to specify a set of processors it may run on.

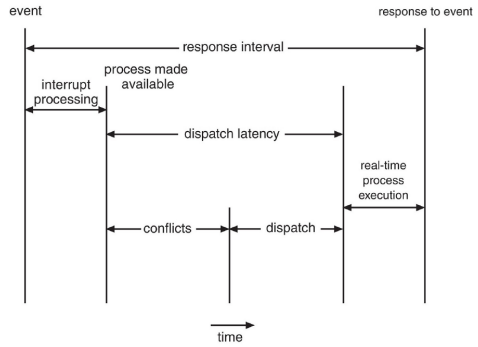
4.10 Real-time systems

Soft real-time systems = Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
Hard real-time systems = task must be serviced by its deadline
Event latency = the amount of time that elapses from when an event occurs to when it is serviced.
Two types of latencies affect performance:

- Interrupt latency = time from arrival of interrupt to start of routine that services interrupt.
- Dispatch latency – time for schedule to take current process off CPU and switch to another



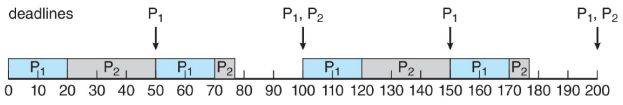
Interrupt latency consists of: 1) determining interrupt type and 2) switching the context.



Conflict phase of dispatch latency is: 1) preemption of any process running in kernel mode and 2) release by low-priority process of of resources needed by high-priority processes.

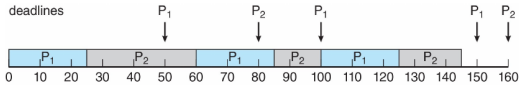
4.10.1 Real-time Priority-based Scheduling

For real-time scheduling, scheduler must support preemptive, priority- based scheduling.
Process may be periodic, have a processing time t, deadline d and period p. Rate of periodic task is $\frac{1}{p}$.
A priority is assigned based on the inverse of its period.



4.10.2 EDF

EDF = Earliest Deadline First Scheduling Priorities are assigned according to deadlines: The earlier the deadline, the higher the priority



4.10.3 Proportional Share Scheduling

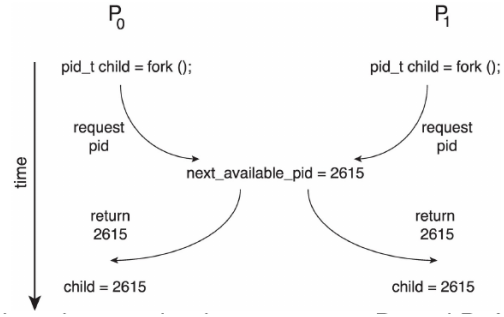
T shares are allocated among all processes in the system
An application receives N shares where $N \leq T$.
This ensures each application will receive $\frac{N}{T}$ of the total processor time.

4.11 Little's formula

n = average queue length W = average waiting time in queue λ = average arrival rate into queue $n = \lambda \cdot W$.

5 Synchronization

A **race condition** is when multiple threads/processes access and manipulate data concurrently, and the result depends on the **order** of the access.
Here, two process acces the same next_pid and create two different children with the same pid.



5.1 Critical section

A critical sections is a segment of instruction in which the process is accessing common variables, tables, writing to files etc. When one process is in a critical section, **no other process may be in their critical section**.
Each process must ask permission to enter the critical section, then exit and continue.

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

5.1.1 Solution criteria

- mutual exclusion - if a process is executing a critical section, no other process will
- progress - If no process is in its critical section and some processes wish to enter, then only those processes not in their remainder section can participate in the decision of which will enter next, and this selection cannot be postponed indefinitely.
- bounded waiting - There is a limit on the number of times other processes are allowed to enter their critical sections after a process has made a request to enter and before that request is granted. (This prevents starvation).

5.1.2 Solutions

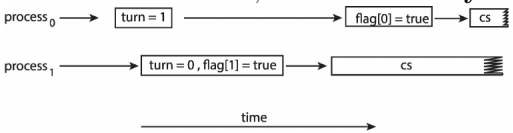
- disable interrupts on entry and enable on exit - risk of starvation, doesn't work with multiple cores
- assume load/store are atomic (cannot be interrupted) - two processes have a turn variable which shows whose turn it is to enter the critical section: doesn't meet progress and bounded waiting reqs.

```
while (true){

    while (turn = = j);

    /* critical section */

    turn = j;

    /* remainder section */

}
```

- **Peterson's solution** - use turn (shows whose turn is it) and a boolean flag (shows if process is ready to enter critical section). When process i is ready, it lets j exectue if its ready, then it executes itself. It doesn't work on modern architectures because compilers may reorder operations that have no dependencies.

```
while (true){

    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j)
        ;

    /* critical section */

    flag[i] = false;

    /* remainder section */

}
```

If the turn and flag setting instructions are changed, we risk getting the processes into the critical section at the same time. Thus, we need **memory barriers**.



A **memory barrier** is an instruction that forces all chances in memory to be propagated to all processors.

Thread 1 now performs

```
while (!flag)
  memory_barrier();
print x
```

Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of x.

For Thread 2 we ensure that the assignment to x occurs before the assignment `flag`.

There exists hardware support for this
- uniprocessors - disable interrupts
- hardware instructions
- atomic variables

5.1.3 Hardware instructions

`test_and_set` - executed atomically, returns the original value of the passed parameter, sets the new value to true.

```
boolean test_and_set (boolean
*target)
    {
        boolean rv = *target;
        *target = true;
        return rv:
    }
```

We can implement a lock (shared variable lock) as such:

```
do {
      while (test_and_set(&lock))
      ; /* do nothing */

          /* critical section */

    lock = false;
          /* remainder section */
 } while (true);
```

`compare_and_swap` - executed atomically, returns the value of original, sets the value = new_value only if value was the expected value.

```
int compare_and_swap(int *value, int expected, int new_value
{
  int temp = *value;
  if (*value == expected)
     *value = new_value;
  return temp;
}
```

We can implement a memory barrier with a shared variable lock initialized to 0.

```
while (true){
          while (compare_and_swap(&lock, 0, 1) != 0)
          ; /* do nothing */

          /* critical section */

          lock = 0;

   /* remainder section */
}
```

Also, we can implement bounded waiting by giving the critical section access to the next waiting process, only unlocking if there is no such process.

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

5.1.4 Atomic variables

Variables of basic data types (int, bool) that have atomic operations on them. For example `sequence` atomic variable and `increment(&sequence)` atomic.

```
    void increment(atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v,temp,temp
        +1));
}
```

5.1.5 Mutex Lock

- boolean variable indicating if lock is available or not.
- protect by acquiring and releasing lock.
- requires busy waiting, it is named also **spinlock**

```
    while (true) {
        acquire lock
        // critical section
        release lock
        // remainder section
}
```

5.1.6 Semaphore Lock

Semaphore S - integer variable indicating available resources.
- `wait()/P()` and `signal()/V()`, both need to be atomic

```
    wait(S) {
        while (S <= 0)
        ; // busy wait
        S--;
}
```

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

A binary semaphore (values 0/1) is a **mutex**.
We can use a **semaphore** to guarantee that two statements S_1 and S_2 in two different processes happen in the correct order.

- Create a semaphore "**synch**" initialized to 0

```
P1:
   S1;
   signal(synch);
P2:
   wait(synch);
   S2;
```

Busy waiting is not a good solution as processes can spend much time in critical zone. We want to implement a waiting queue for ready processes.
We have `block` - place the invoking process in waiting queue, `wakeup` remove one of the processes from the waiting queue and add to ready queue.

```
    typedef struct {
    int value; // value of the semaphore
    struct process *list; // associated queue of
         semaphore, as linked list
} semaphore;
```

The new wait and signal is

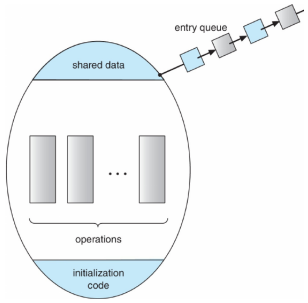
```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

5.1.7 Monitor

- high level synchronization mechanism.
- internal variables accessible only to the code inside the monitor
- only one process may be active inside the monitor

Pseudocode:

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }
    procedure P2 (...) { ... }
    procedure Pn (...) {...}
    initialization code (...) {... }
}
```



5.1.8 Conditional variables

To allow a process to wait within the monitor for a specific event (without busy waiting), we introduce the condition variable.A condition variable x acts as a queue of processes waiting for a specific condition to become true. It does not store a value (unlike a semaphore); it only facilitates waiting and signaling.

Operations The only operations allowed on a condition variable x are:

- `x.wait()` – The process invoking this is suspended and placed in the queue associated with x. Crucially, the process releases the monitor lock, allowing other processes to enter the monitor (potentially to change the condition).
- `x.signal()` – Resumes exactly one suspended process (if any) from x's queue. If no process is waiting, this operation has **no effect** (unlike semaphores, where signal always increments state).

This is a monitor that enforces S1 to happen before S2 using 2 procedures and conditional variables

```
monitor SequenceEnforcer { boolean done; condition
      x;

    procedure F1() {
        S1;
        done = true;
        x.signal();
    }

    procedure F2() {
        if (!done)
            x.wait();
        S2;
    }

    initialization code() {
        done = false;
    }
}
```

Monitor with Semaphore

```
    semaphore mutex; // (initially = 1)
    semaphore next; // (initially = 0)
    int next_count = 0; // number of processes
         waiting inside the monitor

    // procedure P body
    wait(mutex);

    // body of P;

    if (next_count > 0)
        signal(next) // let other process enter
    else
        signal(mutex); // unlock
```

Monitor - Conditional variables

```
    mutex; // initially 1
    // for each condition variable
    semaphore x_sem; // (initially = 0)
    int x_count = 0;

    // wait
    x_count++;
    if (next_count > 0)
        signal(next); // get into the waiting
            queue and unlock the first waiting
            variable
    else
        signal(mutex); // get access over
            resources
    wait(x_sem);
    x_count--;

    // signal
    if (x_count > 0) {
    next_count++;
    signal(x_sem);
        wait(next);
        next_count--;
    }
```

We can add conditional waiting `x.wait(c)` where c is a priority and the process with lowest number (highest priotity) will be accessed next.

5.1.9 Single Resource Allocator

Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource.

```
R.acquire(t); // t time
...
// access the resurce;
...
R.release;
```

Which can be implemented as

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = true;
    }

    void release() {
        busy = false;
        x.signal();
    }
    initialization code() {
        busy = false;
    }
}
```

5.2 Liveliness

Liveness refers to a set of properties that a system must satisfy to ensure processes make progress. Indefinite waiting - liveliness failure.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the 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);`

Consider if P0 executes wait(S) and P1 wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q)

However, P1 is waiting until P0 execute signal(S).

Since these signal() operations will never be executed, P0 and P1 are deadlocked.

Other forms of **deadlock** :

- starvation - A process may never be removed from the semaphore queue in which it is suspended
- Priority inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

6 Synchronization Examples

6.1 Bounded-Buffer Problem

Producer-Consumer problem: producer process produces information that is consumed by a consumer process.

Bounded buffer = Producer must wait if all buffers are full; Consumer waits if there is no buffer to consume.

There are n buffers.

Solution:

- semaphore mutex (binary semaphore, acts as mutex)
- semaphore full (initialized to 0)
- semaphore empty (initialized to n)

```
while (true) {
        ...
        /* produce an item in next_produced */
        ...
        wait(empty);
        wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
        signal(mutex);
        signal(full);
}
```

```
while (true) {
        wait(full);
        wait(mutex);
        ...
        /* remove an item from buffer to
             next_consumed */
        ...
        signal(mutex);
        signal(empty);
        ...
        /* consume the item in next consumed */
        ...
}
```

6.2 Readers and Writers Problem

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

- **Readers** - only read the data set; they do not perform any updates
- **Writers** - can both read and write

Allow multiple readers to read at the same time.

One writer can access the shared data at the same time.

Solution:

- Semaphore rw_mutex initialized to 1
- Semaphore mutex initialized to 1
- Integer read_count initialized to 0

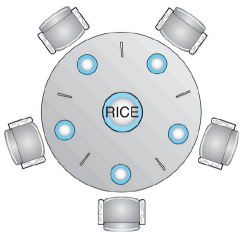
```
// Reader
while (true){
        wait(mutex);
        read_count++;
        if (read_count == 1) /* first reader */
                wait(rw_mutex);
                signal(mutex);
        ...
                /* reading is performed */
        ...
                wait(mutex);
        read_count--;
        if (read_count == 0) /* last reader */
                signal(rw_mutex);
        signal(mutex);
}
```

```
// Writer
while (true) {
        wait(rw_mutex);
        ...
        /* writing is performed */
        ...
        signal(rw_mutex);
}
```

There is a problem, a writer process may never write. This problem above is called **"First reader-writer"**. **"Second reader-writer"**: Once a writer is ready to write, no "newly arrived reader" is allowed to read. On some systems, there are rw_locks specifically designed for this.

6.3 Dining-Philosophers Problem

- N philosophers sit at a round table with a bowel of rice in the middle.
- They spend their lives alternating thinking and eating.
- They do not interact with their neighbors.
- Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl (Need both to eat, then release both when done)



Naive semaphore solution (this results in deadlock, consider everyone picks-up left chopstick):

```
while (true){
        wait (chopstick[i] );
        wait (chopStick[ (i + 1) % 5] );
        /* eat for awhile */
        signal (chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
        /* think for awhile */
}
```

Some deadlock-free solution, though not starvation-free:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- An odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

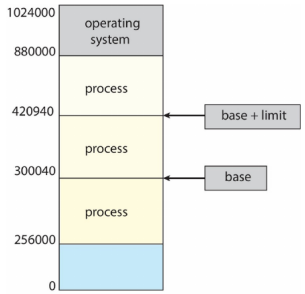
7 Main Memory

The CPU can directly access only **main memory** and **registers**. While register access occurs in one CPU clock cycle or less, main memory access can take many cycles, potentially causing a CPU **stall**. To bridge this speed gap, a **cache** is placed between the CPU and main memory.

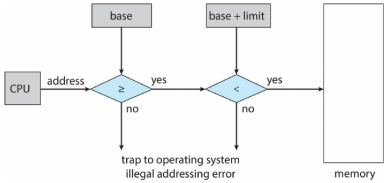
7.1 Memory Protection

To ensure correct operation, we must protect the operating system from access by user processes and protect user processes from one another. This protection is provided by a pair of **base** and **limit registers**.

- **Base register**: Holds the smallest legal physical memory address.
- **Limit register**: Specifies the size of the range (logical address space).

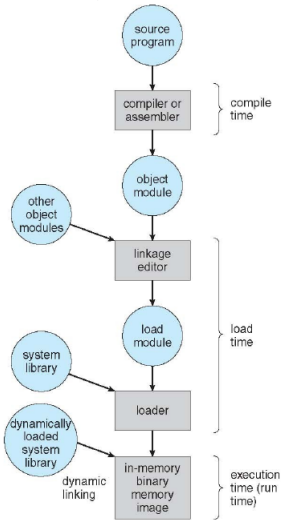


The hardware must check every memory access generated in user mode to ensure it is within the legal range: $base \leq address < base + limit$. If it is outside this range, the hardware traps to the OS with an illegal addressing error.



7.2 Address Binding

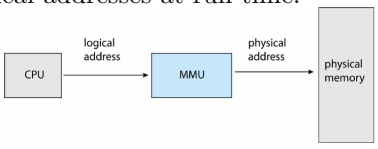
A user program goes through several steps before being executed. Addresses are represented in different ways: symbolic in source code, relocatable in compiled modules, and absolute after linking or loading .



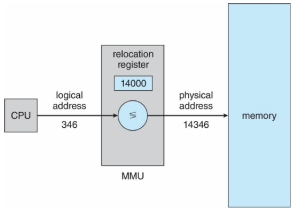
7.3 Logical vs. Physical Address Space

- **Logical address**: Generated by the CPU; also referred to as a virtual address.
- **Physical address**: The actual address seen by the memory unit.

The **Memory-Management Unit (MMU)** is the hardware device that maps virtual addresses to physical addresses at run time.



In a simple relocation-register scheme, the value in the **relocation register** (base) is added to every address generated by a user process. The user program never sees the real physical addresses.

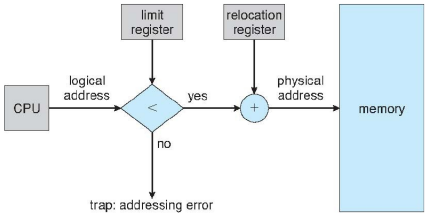


7.4 Dynamic Loading and Linking

- **Dynamic Loading**: A routine is not loaded until it is called. This provides better memory-space utilization as unused routines are never loaded.
- **Dynamic Linking**: Linking is postponed until execution time. A small piece of code called a **stub** is used to locate the appropriate library routine.

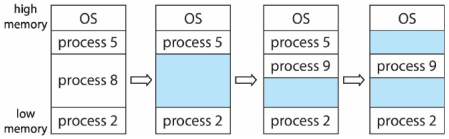
7.5 Contiguous Allocation

In contiguous allocation, each process is contained in a single contiguous section of memory. Memory is usually partitioned into two: one for the resident OS and one for user processes.



7.5.1 Variable Partitioning

The OS tracks which parts of memory are occupied and which are free (**holes**). When a process arrives, it is allocated a hole large enough to accommodate it.



7.5.2 Dynamic Storage-Allocation Problem

To satisfy a request of size n, three common strategies are used:

- **First-fit**: Allocate the first hole that is big enough.
- **Best-fit**: Allocate the smallest hole that is big enough (requires searching the whole list).
- **Worst-fit**: Allocate the largest hole.

First-fit and best-fit are generally better than worst-fit in speed and utilization. The **50-percent rule** suggests that for first-fit, 1/3 of memory may be unusable due to fragmentation.

7.5.3 Fragmentation

- **External Fragmentation**: Total memory exists for a request but is not contiguous. Can be reduced by **compaction**.
- **Internal Fragmentation**: Memory allocated to a process is slightly larger than requested; the difference is unused within the partition.

8 Paging

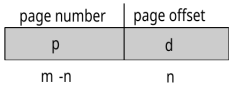
Paging allows the physical address space of a process to be noncontiguous, avoiding external fragmentation.

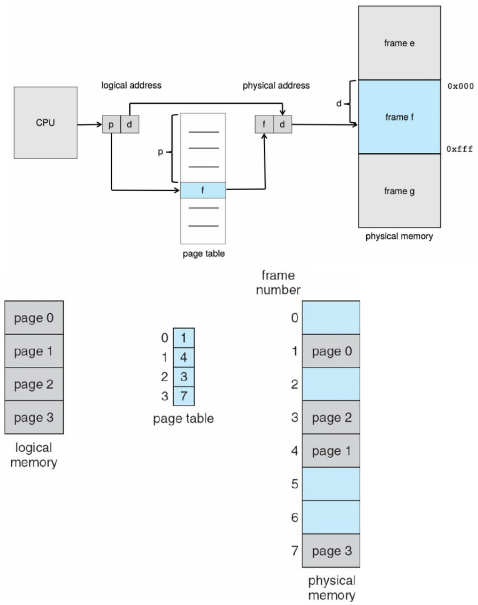
- **Frames**: Fixed-sized blocks of physical memory.
- **Pages**: Blocks of the same size in logical memory.
- **Frame Table**: A data structure used to keep track of all free and occupied frames.

8.1 Address Translation Scheme

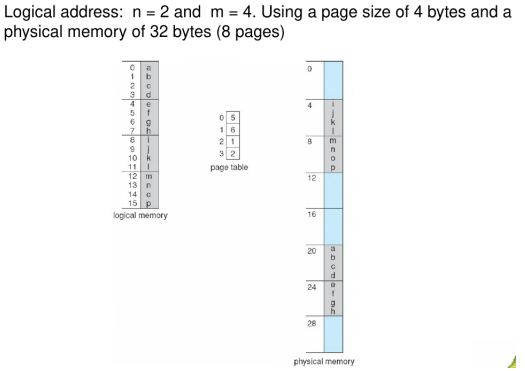
An address generated by the CPU is divided into:

- **Page number (p)**: Used as an index into a **page table**.
- **Page offset (d)**: Combined with the base address from the page table to define the physical address.





In the paging example, logical memory is mapped to physical memory via the page table entries.

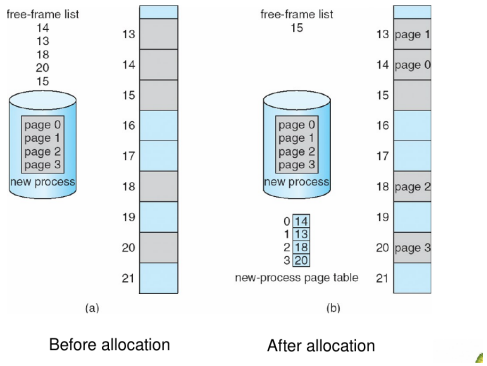


8.2 Calculating Internal Fragmentation

If the page size is 2,048 bytes and a process size is 72,766 bytes, it requires 35 full pages plus 1,086 bytes . The internal fragmentation is $2,048 - 1,086 = 962$ bytes.

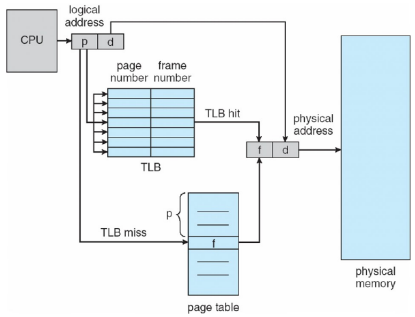
8.3 Free Frames

The OS maintains a free-frame list to allocate memory to new processes.



8.4 TLBs and Performance

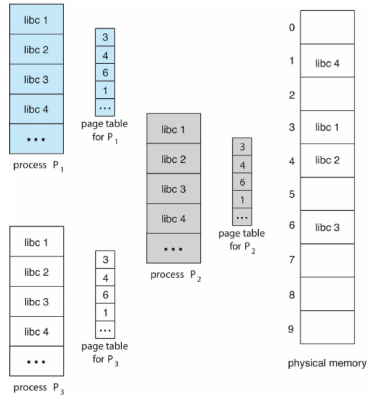
Every data access requires two memory accesses: one for the page table and one for the data. This is solved by using a **Translation Look-aside Buffer (TLB)**, a fast-lookup hardware cache.



Effective Access Time (EAT) calculation example: If memory access is 10ns and hit ratio is 80%: $EAT = 0.80 \times 10 + 0.20 \times 20 = 12$ ns (20% slowdown). With 99% hit ratio: $EAT = 0.99 \times 10 + 0.01 \times 20 = 10.1$ ns (1% slowdown).

8.5 Shared Pages

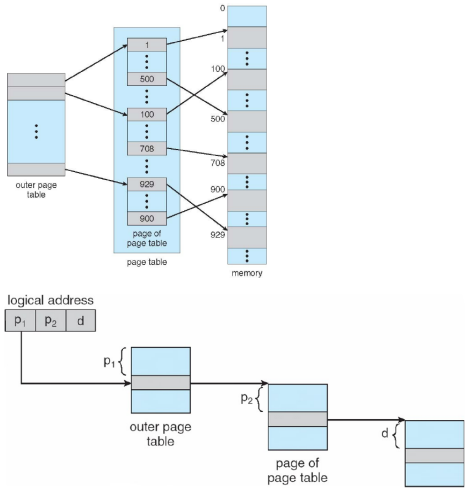
Read-only code (reentrant) can be shared among processes by mapping the same physical frames to different page tables.



9 Page Table Structures

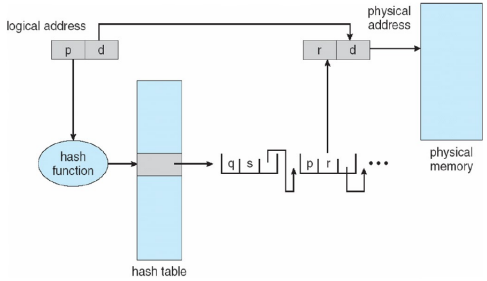
9.1 Hierarchical Page Table

For large address spaces, the page table itself is paged. A two-level scheme divides the logical address into an outer page index (p_1), an inner page displacement (p_2), and the page offset (d).



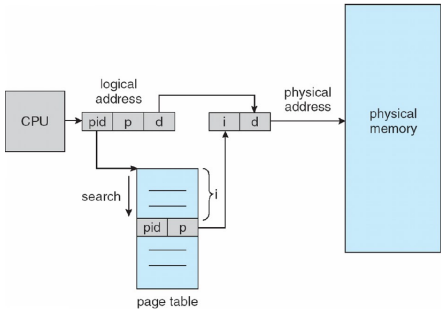
9.2 Hashed Page Table

Common for addresses ¿ 32 bits. The virtual page number is hashed into a table containing a chain of elements that hash to the same location.



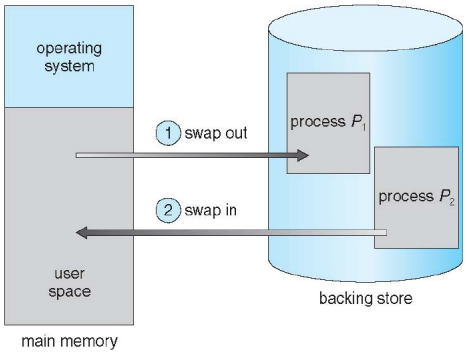
9.3 Inverted Page Table

Rather than one table per process, there is one entry for each real frame of memory. Each entry includes the address and the **PID** of the owning process.



10 Swapping

A process can be swapped out of memory to a **backing store** and later brought back. This allows the total physical memory space of processes to exceed actual physical memory.

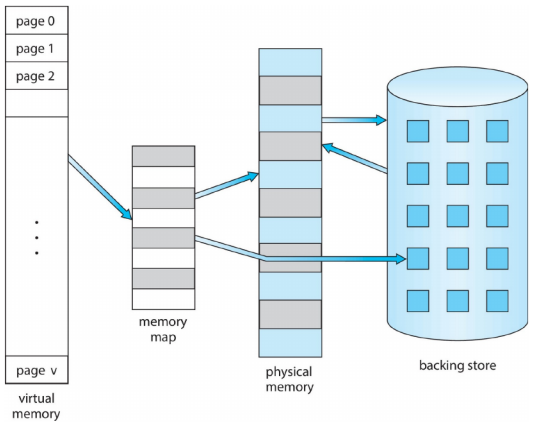


11 Virtual memory

We want to load a program in memory so we can execute it. Main memory may be less than program size (run a 32GB exe on 8GB ram). The entire program code is not needed at the same time

Virtual memory = separation of user logical memory from physical memory

The virtual address space must be mapped to physical address space via a Memory-Mapping unit (MMU).



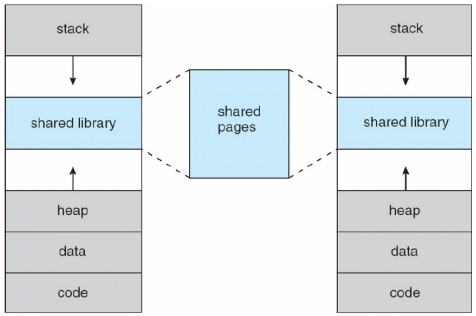
Benefits:

- Only part of the program needs to be in memory for execution
- Logical address space can therefore be much larger than physical address space
- Allows address spaces to be shared by several processes
- More programs running concurrently
- Less I/O needed to load or swap processes

Virtual memory can be implemented via:

- **Demand paging**
- **Demand segmentation**

By virtual memory we can share pages, useful for dynamic loading and linking. Sharing libraries.

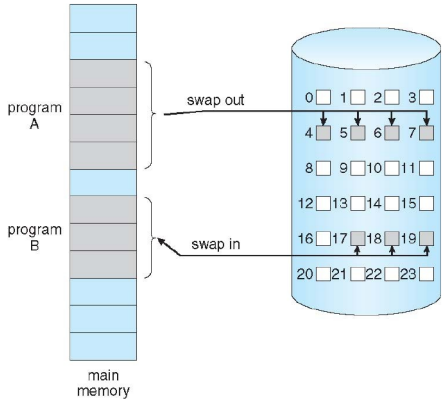


11.1 Demand Paging

Demand paging is a strategy where pages are loaded into memory only when accessed. This results in **lower I/O consumption**, **reduced memory usage**, and **faster response times**.

Lazy Pager: Never swaps a page into memory unless it is specifically needed.

Memory Resident: Pages that are currently located in physical RAM.



11.1.1 Hardware Support

To implement demand paging, the system requires three critical components:

1. **Page Table with Valid-Invalid Bit:**
   - `v`: In-memory (Resident).
   - `i`: Not-in-memory (Page fault or invalid).
2. **Secondary Memory:** A swap device (HDD/SSD) providing **swap space**.
3. **Instruction Restart:** The ability to restart a CPU instruction from scratch after a page fault is resolved.

11.1.2 Handling a Page Fault

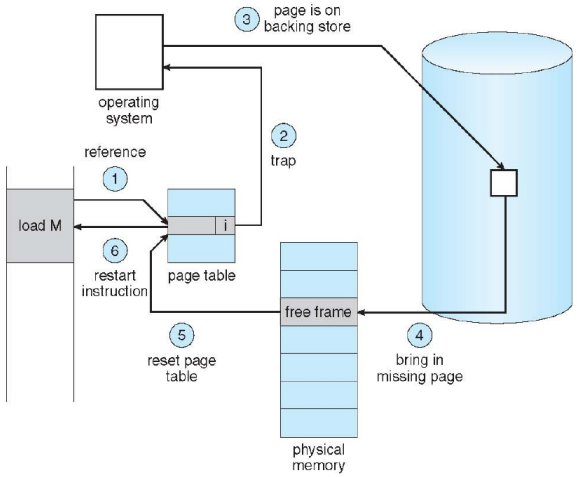
A **Page Fault** occurs when the MMU encounters an 'i' bit during address translation. The OS resolves this via the following sequence:

1. **Trap:** The hardware traps to the OS.
2. **Internal Verification:** OS checks an internal table to see if the reference was a legal access (not-in-memory) or a segment violation (abort).
3. **Allocation:** Locate a frame from the **Free-Frame List**.

4. **Disk I/O:** Schedule a disk operation to read the desired page into the new frame.

5. **Update:** Modify the page table entry to `v` and record the frame number.

6. **Restart:** Resume the instruction that caused the trap.



11.1.3 The Free-Frame List

The OS maintains a pool of available physical frames to satisfy page faults immediately.

- **Zero-fill-on-demand:** Frames are cleared (zeroed) before allocation for security, preventing data leakage between processes.
- **Performance Buffering:** The OS maintains a "low water mark" threshold. If the list shrinks too much, background **Page Replacement** (e.g., LRU) is triggered to reclaim space before the system stalls.

11.1.4 Detailed Execution Stages

1. **Interrupt:** Trap to OS and save process state/registers.
2. **Identify:** Determine the trap was a page fault; verify legality and disk location.
3. **I/O Request:** Issue disk read. While waiting (Seek/Latency), the CPU is context-switched to another user.
4. **Completion:** Receive disk interrupt; save current user state.
5. **Update:** Update Page Table to show the page is now resident.
6. **Resume:** Wait for CPU scheduler, restore original process registers, and restart the instruction.

11.1.5 Performance of Demand Paging

Page Fault Rate (p): $0 \leq p \leq 1$

- If $p = 0$, there are no page faults.
- If $p = 1$, every reference is a fault.

Effective Access Time (EAT):

$$EAT = (1 - p) \times \text{memory access} + p \times (\text{page fault overhead} + \text{swap out/in} + \text{restart})$$

Example:

- Memory access time = **200 ns**
- Average page-fault service time = **8 ms** $(8,000,000$ **ns)**

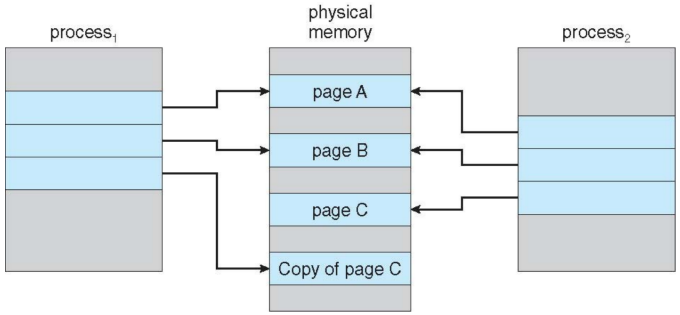
If 1 out of 1,000 accesses causes a page fault ($p = 0.001$):

$$EAT = (1 - 0.001) \times 200 + 0.001 \times 8,000,000$$
$$EAT = 199.8 + 8,000 = \mathbf{8,199.8}\ \text{ns}$$

Conclusion: Even with a 0.1% fault rate, the system slows down by a factor of 40. Performance is highly sensitive to the page fault rate.

11.1.6 Copy-on-Write

Copy-on-Write (COW) = allow both parent and child to initially share the same pages on memory.
If a process modifies a page the page is the copied for that process.
Let's say P_1 and P_2 share pages A, B, C. If P_1 modifies C, then we have:

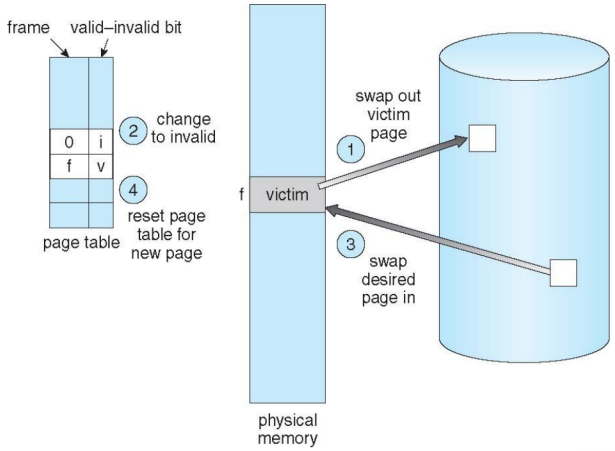


11.2 Page Replacement

If there are no Free Frames available? We need to replace a paged mapped to a frame.
Dirty bit = When a page is selected as a "victim" to be swapped out to make room for a new page, the OS must decide if it needs to write that page back to the disk. Maybe the process changed something and need to save the changes back in memory. If page is modified set dirty bit.
Steps:

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

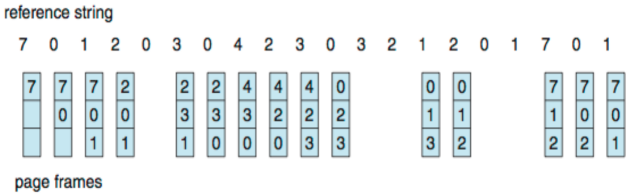
This procedure incereases EAT.



Frame-allocation algorithms = determines how many frames to give each process. Which frames to replace.
Page-replacement algorithms = Want lowest page-fault rate on first access and re-access.

11.2.1 FIFO

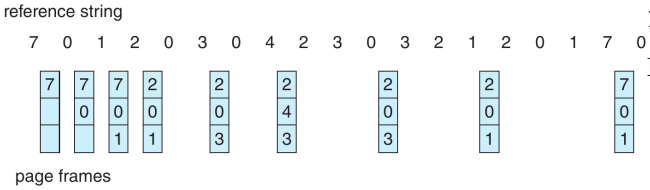
First-In-First-Out (FIFO) First frame in will be the first out when need to replace. Uses a queue.
Belady's Anomaly = adding more frames can cause more page faults.
Example:



11.2.2 OPT (Optimal Algorithm)

- Replace the page that **will not be used for the longest period of time**.
- Problem: We can't read the future. So we need a compromise: LRU.

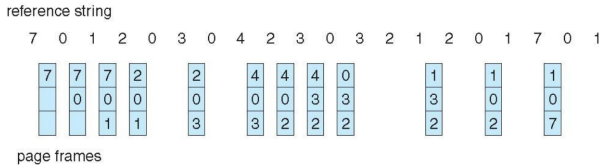
Example:



11.2.3 LRU (Least Recently Used)

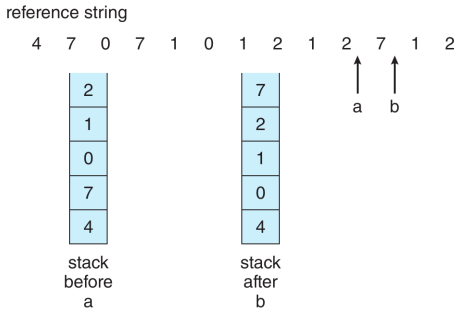
- Use past knowledge rather than future.
- Replace page that **has not been used for the longest period of time**.

Example:



There are two implementations for LRU:

- Counter implementation (Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter; When a page needs to be changed, look at the counters to find smallest value. Search algorithm needed.)
- Stack implementation (Keep a stack of page numbers in a double link form. If a page is referenced move it to the top. No search for replacement. But updates are expensive.)



Because LRU is expensive, we use LRU approximations:

11.2.4 Second-chance algorithm

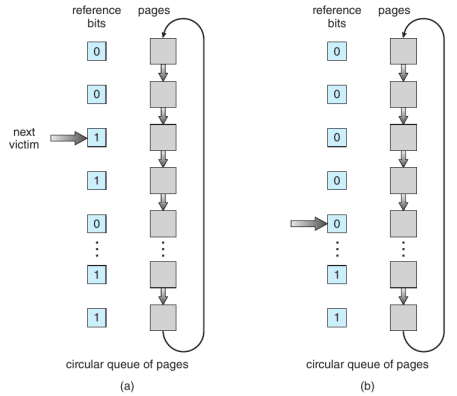
LRU approximation.

- Generally FIFO, plus hardware-provided reference bit
- If page to be replaced has reference bit = 0, replace it, else set bit to 0 and try next page.
- Usually implemented with refernce bit and circular queue.

11.2.5 Enhanced Second-chance Algorithm

We can enhance the second-chance algorithm by considering the reference bit and the modify (dirty) bit as an ordered pair. With these two bits, we have the following four possible classes:

- (0, 0) neither recently used nor modified —best page to replace
- (0, 1) not recently used but modified — not quite as good, because the page will need to be written out before replacement
- (1, 0) recently used but clean—probably will be used again soon
- (1, 1) recently used and modified—probably will be used again soon, and the page will be need to be written out to secondary storage before it can be replaced



11.2.6 Counting Algorithms

Keep a counter of the number of references that have been made to each page.

- **Lease Frequently Used (LFU) Algorithm**: Replaces page with smallest count.
- **Most Frequently Used (MFU) Algorithm**:Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

11.2.7 Page Buffering Algorithms

Make use of free-frame list.
Ideas:

- Maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to secondary storage. Its modify bit is then reset.
- Possibly, keep free frame contents intact and note what is in them. If referenced again before reused, no need to load contents again from disk.

11.3 Frame allocation algorithms

- Each process needs minimum number of frames
- Maximum of course is total frames in the system
- There are two major allocation schemes: **fixed allocation** and **priority allocation**.

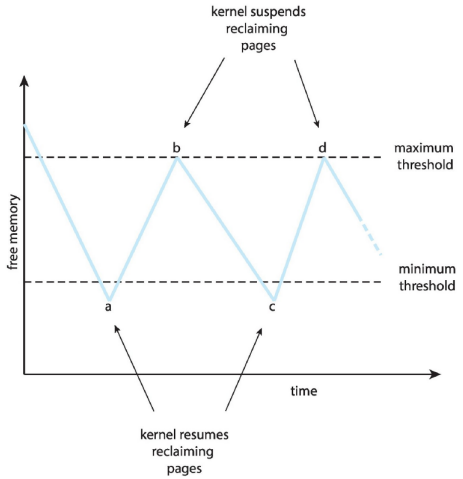
11.3.1 Fixed allocation

Equal allocation: For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames. (keep some for free frame list/pool as well).
Proportional allocation: Allocate according to the size of process (Dynamic as degree of multiprogramming, process sizes change).
s_i = size of process p_i
$S = \sum s_i$
m = total number of frames
a_i = allocation for $p_i = \frac{s_i}{S} \cdot m$
There are two types of locality for allocation:

- **Global replacement**: process selects a replacement frame from the set of all frames; one process can take a frame from another
- **Local replacement**: each process selects from only its own set of allocated frames

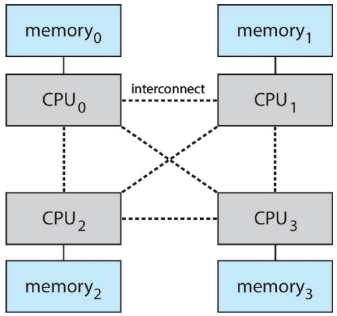
11.3.2 Reclaiming Pages

A strategy to implement global page-replacement policy. All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement. Page replacement is triggered when the list falls below a certain threshold.



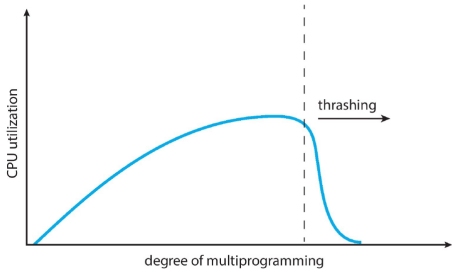
11.3.3 NUMA (Non-uniform memory access)

So far, we assumed that all memory accessed equally Many systems are NUMA - speed of access to memory varies.
Try to allocate memory close to the CPU on which the thread is scheduled.



11.4 Thrashing and Locality

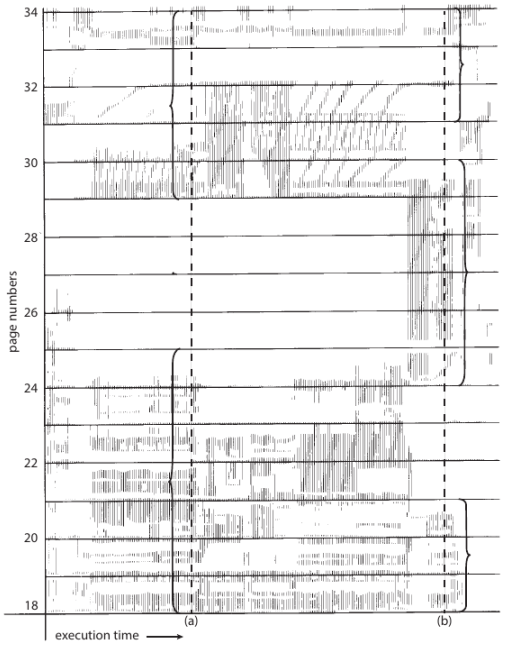
Thrashing = If a process does not have "enough" pages, the page-fault rate is very high. A process is busy swapping pages in and out



Programs do not access memory randomly. Instead, they move through different **localities**. A locality is a set of pages that are actively used together at a specific point in time.

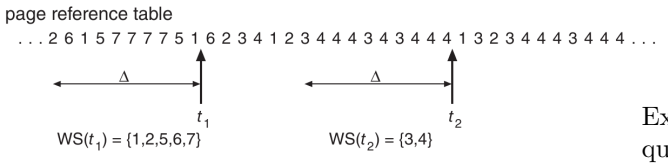
- Temporal Locality: If a page is referenced, it is likely to be referenced again soon (e.g., inside a while loop).
- Spatial Locality: If a page is referenced, nearby pages are likely to be referenced soon (e.g., elements in an array).
- Migration: As a process executes, it migrates from one locality (like an initialization function) to another (like the main processing loop. Localities can overlap (e.g., a function call within a loop).

Locality is the reason that demand paging works.

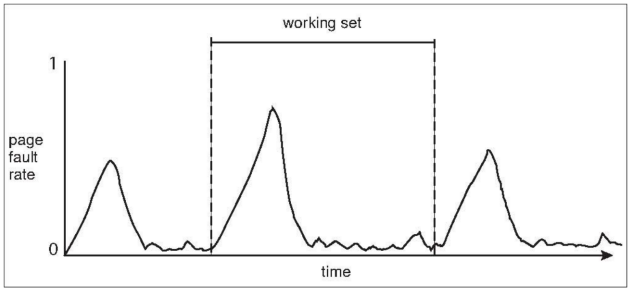


11.4.1 Working-Set Model

The **working-set model** is based on the assumption of locality. This model uses a parameter, Δ, to define the working-set window. The idea is to examine the most recent Δ page references. Example 10,000 instructions.
WSS_i (working set size of process P_i) = total number of pages referenced in the most recent Δ.
$D = \sum WSS_i$ = total demand frames by the system.
m = number of available frames.
If $D > m$ then we get thrashing. You can enforce a policy that if $D > m$, suspend or swap out one of the processes.

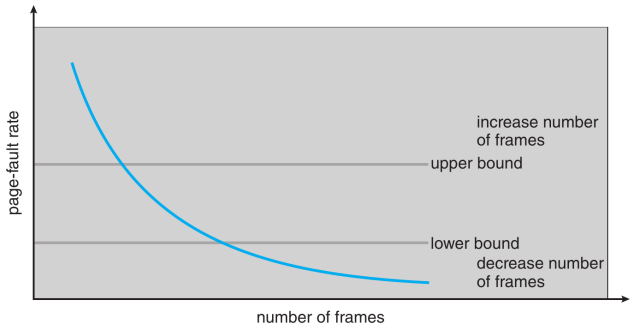


Direct relationship between working set of a process and its page-fault rate. Working set changes over time:



11.4.2 Page Fault Rates

The working-set model is successful, and knowledge of the working set can be useful for prepaging, but it seems a clumsy way to control thrashing. A strategy that uses the page-fault frequency (PFF) takes a more direct approach.



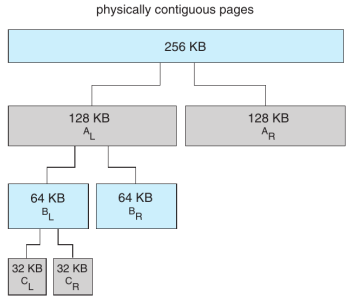
11.4.3 Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages.
- Memory allocated using power-of-2 allocator (Request rounded up to next highest power of 2)
- When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2.

Example: Assume 256KB chunk available, kernel requests 21KB:

- Split in buddies A_l and A_r of 128KB each. One further divided in B_l and B_r of 64KB each. One further divided in C_l and C_r of 32KB each - one used to satisy request.

Coalescing = efficiently combine budies to form larger segments.

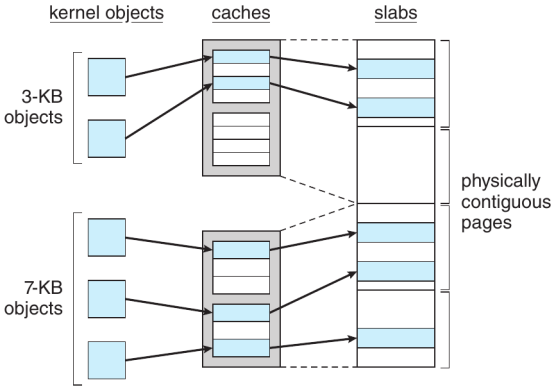


11.4.4 Slab Allocation

Alternate strategy for allocaing kernel memory:

- Slab is one or more physically contiguous pages
- Cache consists of one or more slabs
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab. If no empty slabs, new slab allocated

Benefits include no fragmentation, fast memory request satisfaction



11.5 Prepaging

Pure demand paging means large number of page faults when a process is started.
Prepaging = bring some or all of the pages that will be needed into memory at one time. But if prepaged pages are unused, I/O and memory was wasted
s = pages that are prepaged
α = fraction of prepaged pages used, $0 \leq \alpha \leq 1$
How does the cost of prepaging $s * \alpha$ pages compare to $s * (1 - \alpha)$

11.6 Page size

Sometimes OS designers have a choice. Always power of 2. On average, growing over time
Page size selection must take into consideration:

- Fragmentation
- Page table size
- Resolution
- I/O overhead
- Number of page faults
- Locality
- TLB size and effectiveness

11.7 TLB Reach

TLB (Translation Lookaside Buffer) = small cache located within the CPU's Memory Management Unit (MMU). Think of the TLB as a fast-lookup table that maps Virtual Page Numbers directly to Physical Frame Numbers. Ideally, working set is stored in the TLB.

- **TLB Hit:** The CPU finds the translation in the TLB.
- **TLB Miss:** The CPU finds the translation in the TLB.
- **TLB Reach:** The amount of memory accessible form the TLB = (TLB Size) X (Page Size)

Effective acces time changes:

$$EAT = \alpha(\epsilon + ma) + (1 - \alpha)(\epsilon + 2ma)$$

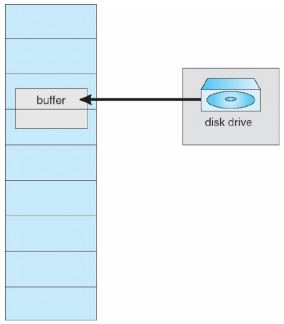
11.8 Program Structure

System performance can be improved if the user (or compiler) has an awareness of the underlying demand paging.

```
\\This has 128 x 128 = 16,384 page faults
for (j = 0; j <128; j++)
        for (i = 0; i < 128; i++)
                data[i,j] = 0;

\\This has 128 page faults
for (j = 0; j <128; j++)
        for (i = 0; i < 128; i++)
                data[i,j] = 0;
```

11.9 I/O Interlock

I/O Interlock = Pages must sometimes be locked into memory.
Pinning = pin pages to lock into memory



12 File-System Interface

A **file** is a contiguous logical address space defined by its creator, containing data (numeric, character, binary) or programs .

12.1 File Attributes and Operations

File Attributes are stored in the directory structure on disk and vary by system :

- **Name:** Human-readable identifier.
- **Identifier:** Unique tag (number) within the file system.
- **Type:** Support for different formats (e.g., .exe, .txt).
- **Location:** Pointer to the device and location on that device.
- **Size:** Current file size.
- **Protection:** Controls for reading, writing, and executing.
- **Time, date, and user ID:** Data for security and usage monitoring.

12.1.1 File Operations

The operating system provides several basic system calls to manage files :

- **Create / Delete / Truncate**
- **Read / Write:** Handled at specific pointer locations.
- **Seek:** Repositioning the pointer within the file.
- **Open(F_i):** Searches the directory for entry F_i and copies it to memory.
- **Close(F_i):** Moves memory content back to the directory structure.

12.1.2 File Locking

Used to mediate access to a file, similar to reader-writer locks :

- **Shared Lock:** Multiple processes can acquire concurrently (reader lock).
- **Exclusive Lock:** Only one process can hold the lock (writer lock).
- **Advisory vs. Mandatory:** In advisory, processes check lock status; in mandatory, the OS enforces access denial.

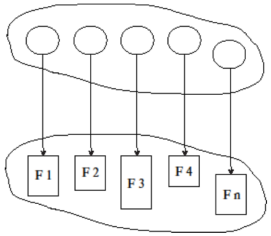
12.2 Open Files

To manage open files, the OS maintains an **open-file table** . Key data pieces include:

- **File Pointer:** Per-process tracking of the last read/write location.
- **File-open Count:** Tracks how many processes have the file open to determine when to remove it from the table.
- **Disk Location:** Cached information for fast data access.
- **Access Rights:** Mode information (read-/write) for each process.

12.3 Directory Structure and Operations

The directory is a collection of nodes containing information about all files .



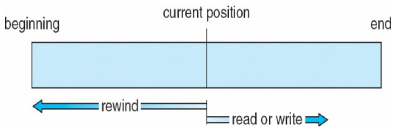
Operations performed on directories: Search, Create, Delete, List, Rename, and Traverse the system .

12.4 Access Methods

12.4.1 Sequential Access

Information is processed in order, one record after the other .

- **Operations:** `read next`, `write next`, `reset`.



12.4.2 Direct Access

Allows fixed-length logical records to be read or written in any order by referring to a **relative block number** (n) .

- **Operations:** `read n`, `write n`, `position to n`.

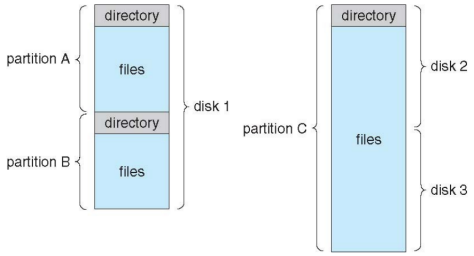
12.4.3 Simulating Sequential on Direct Access

Sequential access can be simulated by maintaining a current position pointer (cp) :

- **Reset:** $cp = 0$;
- **Read Next:** `read cp; cp = cp + 1;`

12.5 Disk and File System Organization

Disks are subdivided into **partitions** (or slices) .

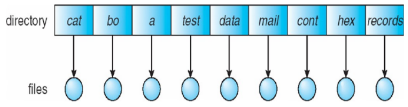


12.6 Directory Logical Structures

12.6.1 Single-Level Directory

All files are contained in the same directory for all users .

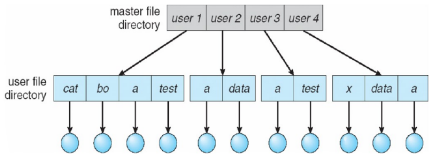
- **Pros:** Simple.
- **Cons:** Naming conflicts (two users can't use the same name) and no grouping.



12.6.2 Two-Level Directory

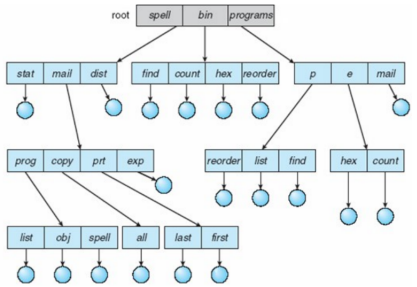
Each user has their own **user file directory** (UFD) under a **master file directory** (MFD) .

- **Pros:** Solves naming conflicts; efficient searching via path names.
- **Cons:** Still lacks sophisticated grouping capability.



12.6.3 Tree-Structured Directories

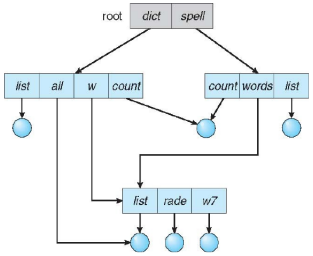
The most common structure, allowing users to create subdirectories and organize files arbitrarily .



12.6.4 Acyclic-Graph Directories

Allows shared subdirectories and files, meaning a file can exist in two directories simultaneously (aliasing) .

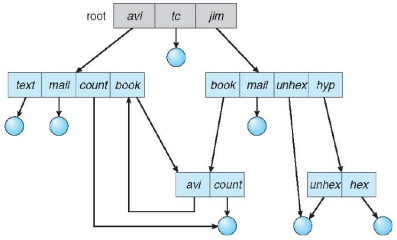
- **Implementation:** Uses **Links** (pointers to existing files).
- **Deletion:** Can lead to dangling pointers. Solutions include **backpointers** or **reference counts** (only delete file when count = 0).



12.6.5 General Graph Directory

Allows cycles within the directory structure .

- **Cycle Detection:** To avoid infinite loops during traversal, systems may use **garbage collection** or a **cycle detection algorithm** every time a link is added.



12.7 Protection

The file owner must control what can be done and by whom . **Access Types:** Read, Write, Execute, Append, Delete, List .

12.7.1 UNIX RWX Access

In UNIX/Linux, protection is handled via three classes of users with three bits (Read: first bit, Write: second bit, eXecute: third bit) :

1. **Owner:** (e.g., 7 → 111)
2. **Group:** (e.g., 6 → 110)
3. **Public:** (e.g., 1 → 001)

Commands like `chmod 761 game` define these permissions, while `chgrp` attaches a specific group to a file .

13 File-system Implementation

13.1 File System Structure and Layers

The file system resides on secondary storage and provides a user interface to storage, mapping logical units to physical blocks. It is organized into layers to reduce complexity:

- **Application Programs**: User-level requests.
- **Logical File System**: Manages metadata and directory structures; translates file names into file numbers/handles using File Control Blocks (FCB) - inodes in UNIX.
- **File-Organization Module**: Translates logical block numbers to physical block numbers and manages free space.
- **Basic File System**: Issues generic commands to device drivers and manages memory buffers and caches.
- **I/O Control**: Contains device drivers and interrupt handlers to bridge the gap between high-level commands and hardware.
- **Devices**: Physical hardware like disks or NVM.

13.2 Control Blocks

13.2.1 Boot Control Block

Contains the information needed by the system to boot the OS from a specific volume. It is typically the first block of an OS-containing volume.

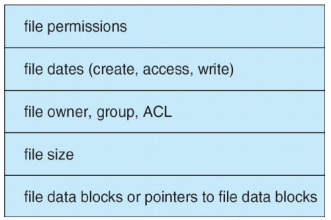
13.2.2 Volume control block (superblock, master file table)

Contains volume details: Total # of blocks, # of free blocks, block size, free block pointers or array.

13.2.3 File Control Block (FCB)

An OS maintains an FCB per file, containing metadata such as:

- Permissions (read/write/execute).
- File dates (creation, access, write)
- Owner, group, and Access Control List (ACL)
- File size and data block pointers

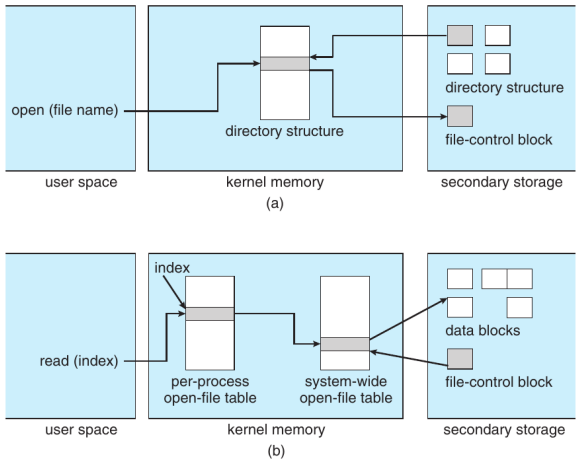


13.3 In-Memory Structures

The OS uses in-memory structures to track active file systems and open files:

- **Mount Table**: Stores information about mounted volumes and their types.
- **System-wide Open-file Table**: Contains a copy of the FCB for every file open in the system.
- **Per-process Open-file Table**: Tracks the current file pointer and points to the system-wide table.

a) Refers to opening a file.
b) Refers to reading a file.



13.4 Directory Implementation

- **Linear List**: A simple list of names with pointers to data blocks; easy to program but requires linear search time ($O(n)$).
- **Hash Table**: Uses a hash function to reduce search time, though it must handle collisions.

13.5 Allocation Methods

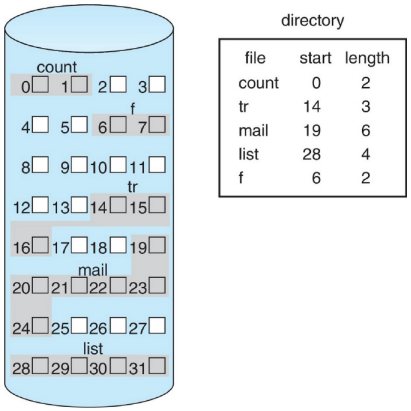
An allocation method refers to how disk blocks are allocated for files.

13.5.1 Contiguous Allocation

Each file occupies a set of contiguous blocks on the disk.

- **Benefits**: Best performance for sequential and random access.
- **Logic**: Needs only starting block number and length.
- **Issues**: External fragmentation and difficulty predicting file growth.

Example:
Mapping from logical to physical (block size = 512 bytes). Block to be accessed = starting address + Q. Displacement into block = R.



13.5.2 Extent-Based Systems

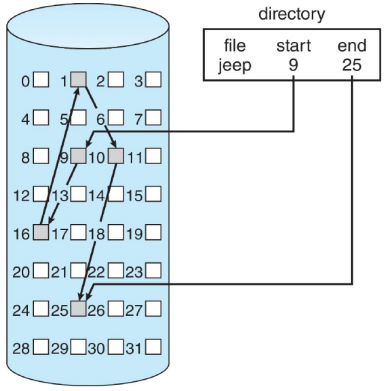
Extent-based file systems allocate disk blocks in extents. An extent is a contiguous block of disks. Extents are allocated for file allocation. A file consists of one or more extents.

13.5.3 Linked Allocation

Each file is a linked list of blocks; pointers are stored within the blocks themselves. File ends at NULL.

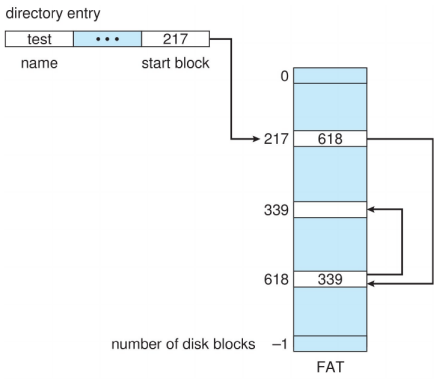
- **Benefits**: No external fragmentation.
- **Issues**: Not suitable for random access; a single lost pointer breaks the file.

Example: Mapping from logical to physical (block size = 512 bytes). Block to be accessed is the Q th block in the linked chain of blocks representing the file.
Displacement into block = R + 1



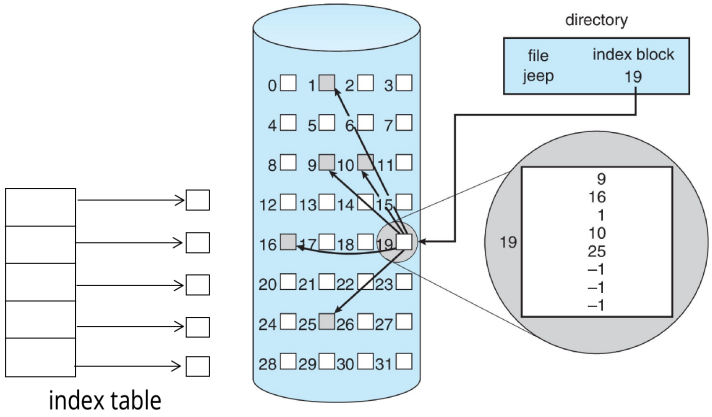
13.5.4 File Allocation Table (FAT)

A variation of linked allocation where all pointers are stored in a central table at the beginning of the volume. This table is cacheable, making access faster than standard linked allocation.



13.5.5 Indexed Allocation

Each file has its own **index block** containing pointers to its data blocks. This allows efficient random access without external fragmentation.



13.6 Two-Level Indexed Scheme

For very large files, a single index block may be insufficient. A **two-level index** uses an outer-index pointing to inner-index blocks, which then point to the data blocks.

- **Example**: Using 4KB blocks and 4-byte pointers, an outer index can store 1,024 pointers to inner indices, supporting file sizes up to 4GB.

13.7 Free-Space Management

13.7.1 Bitmap (Bit Vector)

The disk is represented by a sequence of bits: 1 indicates a free block, 0 indicates occupied.

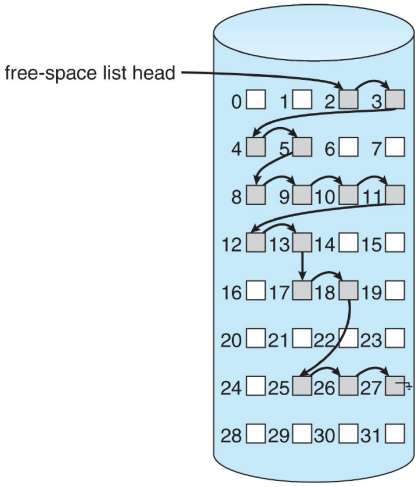
- **Calculation of Bitmap Size**: If disk size is 2^{40} bytes (1TB) and block size is 2^{12} bytes (4KB):

$$n = \frac{2^{40}}{2^{12}} = 2^{28} \text{ bits}$$

$$\text{Size in bytes} = \frac{2^{28}}{8} = 2^{25} \text{ bytes} = 32\text{MB}$$

13.7.2 Linked list (free list)

Another approach to free-space management is to link together all the free blocks, keeping a pointer to the first free block in a special location in the file system and caching it in memory.



- Cannot get contiguous space easily.
- No waste. Linked Free Space List on Disk of space.
- No need to traverse the entire list (if # of free blocks recorded)

Other approaches to improve the free list include:

- **Grouping**: Modify linked list to store address of next n-1 free blocks in first free block, plus a pointer to next block that contains free-block pointers.
- **Counting**: Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering. Keep address of first free block and count of following free blocks. Free space list then has entries containing addresses and counts.
- **Space Maps (ZFS)**: A space map is essentially an append-only list of events (allocations and frees). When ZFS needs to find space, it reads this compact log into memory and constructs an AVL tree (a balanced binary search tree) to represent the free segments.

13.8 Efficiency and Performance

- **Buffer cache** = separate section of main memory for frequently used blocks.
- **Free-behind and read-ahead** – techniques to optimize sequential access.
- **Asynchronous writes** = buffer-able, faster

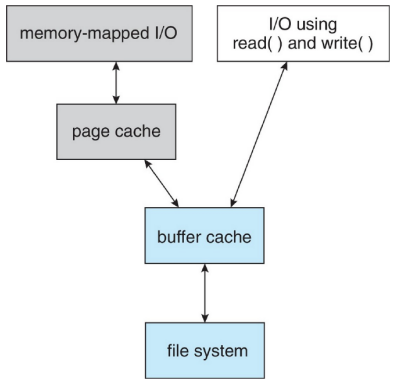
13.8.1 Page Cache

A page cache caches pages rather than disk blocks using virtual memory techniques and addresses.
Memory-mapped I/O uses a page cache.

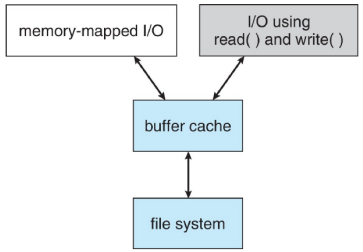
13.8.2 Caching vs. Unified Buffer Cache

- **Standard I/O**: Uses a buffer (disk) cache for file system blocks.
- **Memory-Mapped I/O**: Uses a page cache for virtual memory pages.
- **Unified Buffer Cache**: A single cache for both memory-mapped and standard I/O, preventing "double caching" and improving efficiency.

I/O Without a Unified Buffer Cache:



I/O Using a Unified Buffer Cache:



13.8.3 Recovery

Consistency checking = compares data in directory structure with data blocks on disk, and tries to fix inconsistencies

13.9 Log Structured File Systems

Log structured (or journaling) file systems record each metadata update to the file system as a transaction.
All transactions are written to a log:

- A transaction is considered committed once it is written to the log (sequentially).
- Sometimes to a separate device or section of disk.
- However, the file system may not yet be updated.

The transactions in the log are asynchronously written to the file system structures. When the file system structures are modified, the transaction is removed from the log.
Data safety: If the file system crashes, all remaining transactions in the log must still be performed. Faster recovery from crash, removes chance of inconsistency of metadata

13.10 In-Memory File System Structures

- **Mount table** storing file system mounts, mount points, file system types.
- **System-wide open-file table** contains a copy of the FCB of each file and other info.
- **Per-process open-file table** contains pointers to appropriate entries in system-wide open-file table as well as other info.

14 Storage Units and Powers of Two

Power of 2	Exact Value (Bytes)	Abbreviation	Terminology
2^{10}	1,024	KB	Kilobyte
2^{20}	1,048,576	MB	Megabyte
2^{30}	1,073,741,824	GB	Gigabyte
2^{40}	1,099,511,627,776	TB	Terabyte
2^{50}	1,125,899,906,842,624	PB	Petabyte