Operating Systems

LISHA/UFSC

guto@lisha.ufsc.br
http://www.lisha.ufsc.br/~guto

May 13, 2016

Computer Systems

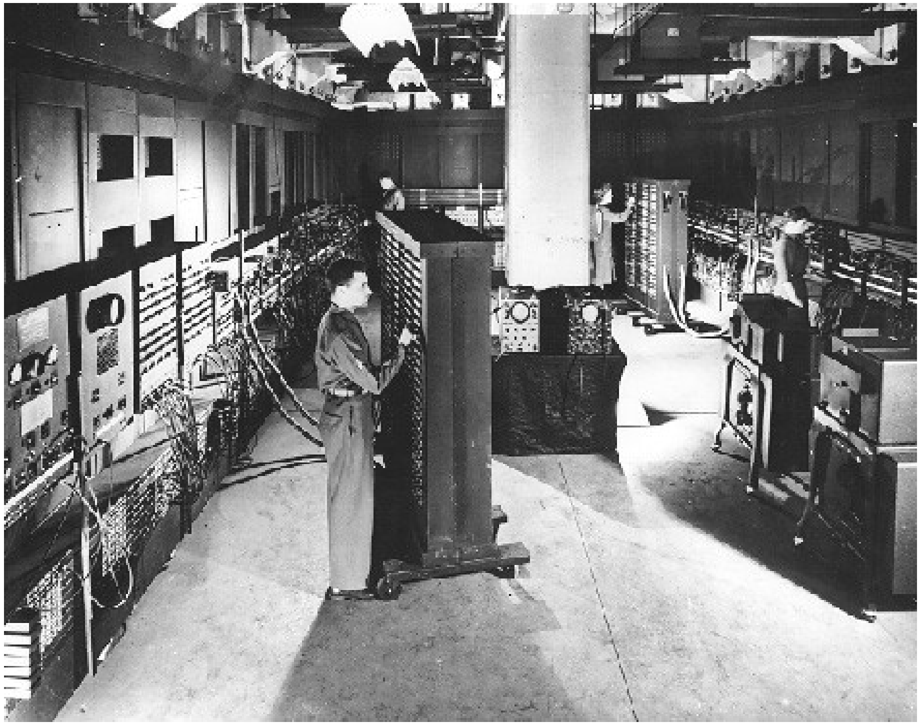
- Hardware
  - CPU + memory + I/O devices
- Operating system
- Application programs
  - Actual goal of computer systems
  - Databases, automation, games, etc
- Users
  - Define computing problems to be solved
  - People, machines, other computers

Operating Systems

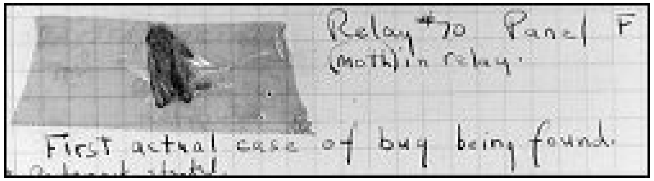
- Virtual machine perspective
  - OS extends the hardware as to implement a higher-level interface to applications
- Resource manager perspective
  - OS manages system resources (processors, memory, disk, etc) for applications' convenience

Historic Perspective

- First generation (1945 - 1955)
  - Vacuum tube
  - No software at all
  - Operated through cable switches



ENIAC (1946)



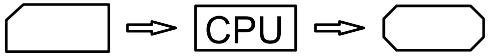
Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

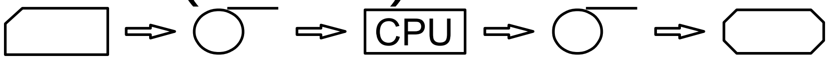
First bug 'caught' by
Grace Murray Hopper, 1945.

Historic Perspective

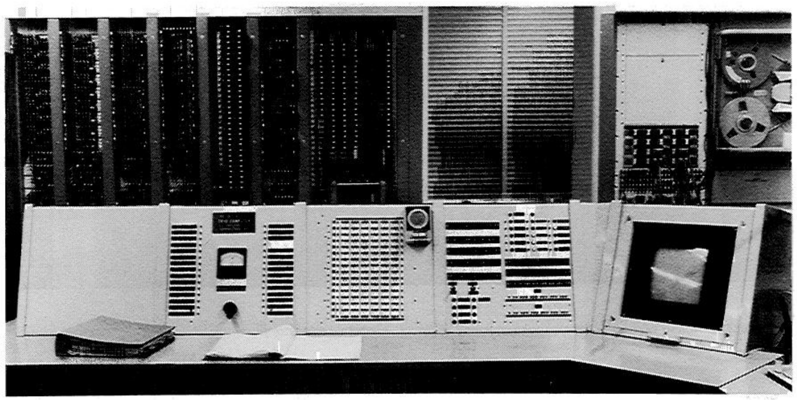
- Second generation (1955 - 1965)
  - Transistor
  - Device drivers
  - First programming languages (Fortran)
  - Monitor (punched card reader)



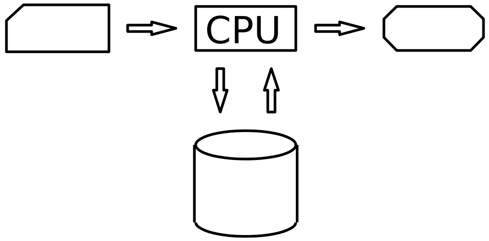
  - Batch (off-line)



    - Spooler (*Simultaneous Peripheral Operation On-Line*)



TX-0 Transistorized Experimental Computer (1956)



Historic Perspective

- Third generation (1965 - 1980)
  - Integrated circuit (TI IC)
  - First generic OOSS (IBM OS/360)
  - Multiprogramming (CPU/IO overlap)
  - Time-sharing (MIT CTSS)
  - MULTICS (MIT, BELL, GE)
  - PDP-11 (DEC)
  - UNIX (BELL)

PDP-11/20 (1970)

Historic Perspective

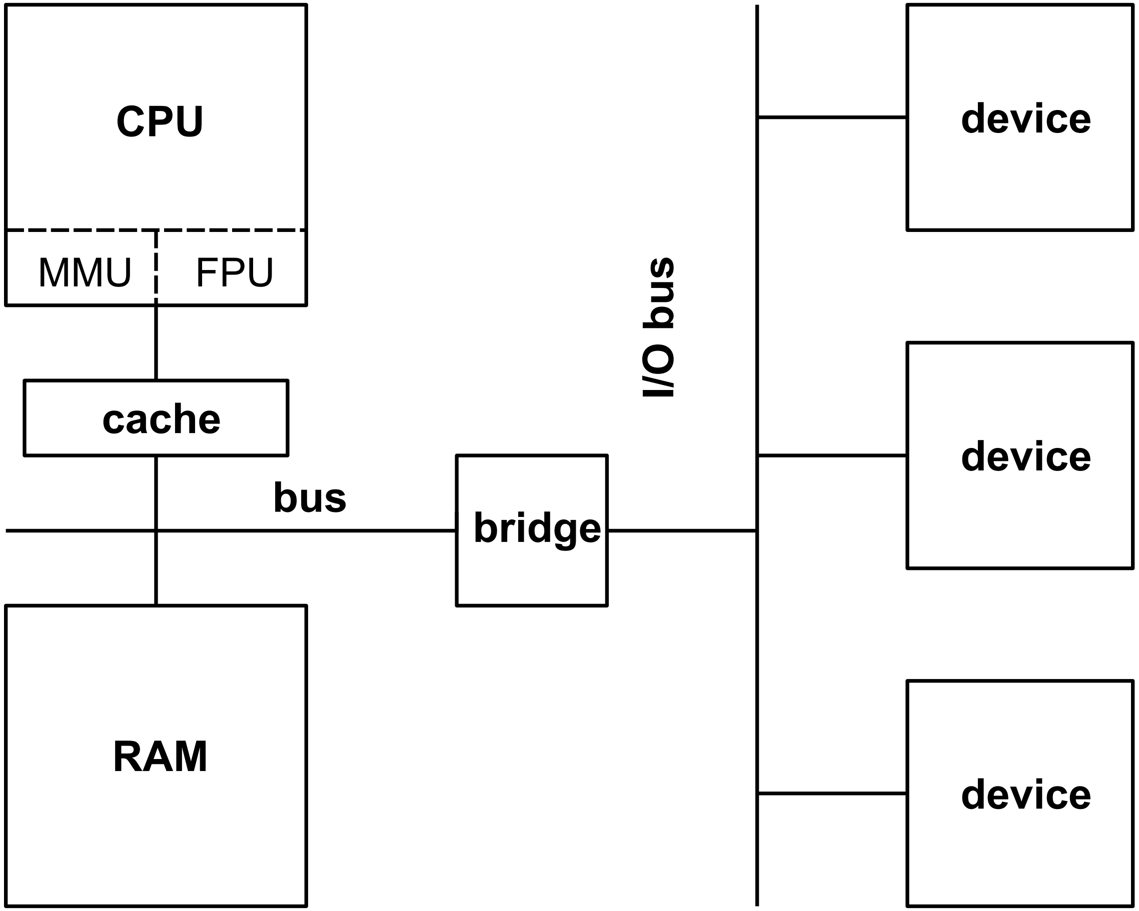
- Fourth generation (1980 - ?)
  - Microprocessor
  - MS-DOS, UNIX
  - Network systems
  - Distributed systems
  - Real-time systems

Apple MacIntosh SE/30
(1972)

Historic Perspective

- Fifth generation (?)
  - Hardware
    - Parallel?
    - Embedded?
    - Ubiquitous!
  - Software
    - Human interface!
    - Artificial intelligence???

A Typical Computer



Computer System Structures

- Motivation
  - Overlap CPU and I/O operations to improve performance
  - Avoid inter-process interference
- Interrupts
  - Avoids busy-waiting
  - I/O device receives a service request and generates and interrupt when the request has been accomplished
  - Transparent to processes
- Direct Memory Access (DMA)
  - Data transfer between I/O device and main memory without CPU assistance

Computer System Structures

- Resource protection
  - Enable the OS to define policies
  - Violations causes a *trap* into the OS
- CPU
  - Dual-mode operation
    - Supervisor mode: whole instruction set, restricted to the OS
    - User mode: unprivileged instruction subset (e.g. no I/O)
  - Timer
    - Timer interrupts periodically transfer control to OS
- Memory
  - Memory Management Unit (MMU)
    - OS isolation
    - Private address spaces for each process
    - I/O device controllers' registers protection

Operating System Services

- Process management
  - Creation and destruction of processes
  - Resource allocation and reclaiming
  - CPU scheduling (and process accounting)
  - Process synchronization
  - Process communication
  - Deadlock handling
- Memory management
  - Memory allocation and deallocation
  - Integrity maintenance (what belongs to whom)
  - Swapping
  - Virtual memory

Operating System Services

- I/O management
  - Buffering/caching
  - Scheduling (e.g. disk, network)
  - Device drivers
- File management
  - Creation, manipulation, and deletion of files
  - Creation, manipulation, and deletion of directories
  - Mapping of files onto disks
- Networking
  - Routing, contention, and security of messages
  - Heterogeneity
  - User interface

Operating System Services

- Protection
  - Access control to resources
  - Logging
  - Procedure validation
- Interface
  - OS provides services to applications by means of an Application Program Interface (API)
    - If the OS is in a different protection domain than applications (e.g. kernel), a system call mechanism is used
  - User interaction
    - Command interpreter (*shell*)

Operating System Architectures

- Monolithic
  - The whole OS comprises a single complex program that is responsible for all services
- Virtual machine
  - OS services are delivered as a private virtual machine to each application process
- Kernel + servers
  - Crucial OS parts, responsible for fundamental services, are kept in a protected kernel
  - Advanced services are delegated to servers that run as ordinary processes

Operating System Architectures

- Microkernel + servers
  - Only services needed to support server processes are kept in the kernel
- Exokernel + libraries
  - Physical resources (CPU, memory, cache) are safely exported to be handled directly by application programs
  - Libraries provide typical implementations for OS services
- Embedded into the application
  - Usually deployed with single-application systems
  - Only OS services required by the application get linked with it

Operating System Engineering

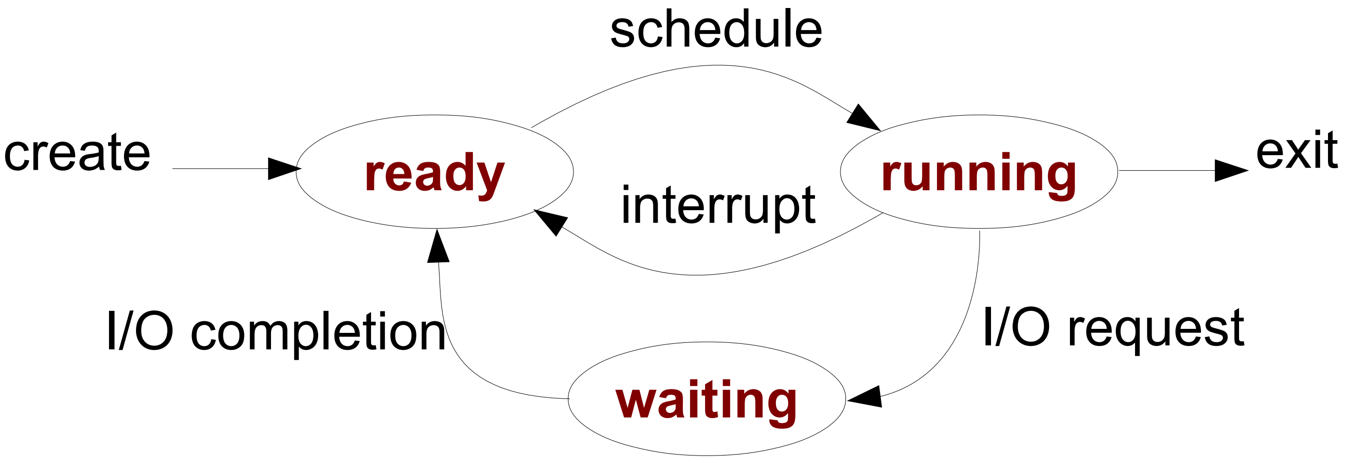
- Structured
  - OS is decomposed in a set of procedures
  - Modifications require the whole system to be rebuilt
- Modular
  - OS is decomposed in a set o modules (e.g. subsystems, service class, etc)
  - Enables replugging
- Object-oriented
  - Similar to modular, but using more powerful software engineering techniques
- Component-based
  - OS is decomposed in a set of reusable components (public interface accesses only)

Process Management

- Process
  - Is a running program
  - Is an active entity
  - Has context and state
  - Is sequentially executed
    - A single instruction is executed on behalf of a process at any time
  - Also called
    - Job on batch systems
    - Task on time-sharing systems

Process State

- Process state
  - Running: process' instructions are being executed
  - Waiting: process is waiting for some event (e.g. I/O completion)
  - Ready: process is ready for execution, but must waiting for a processor to become available

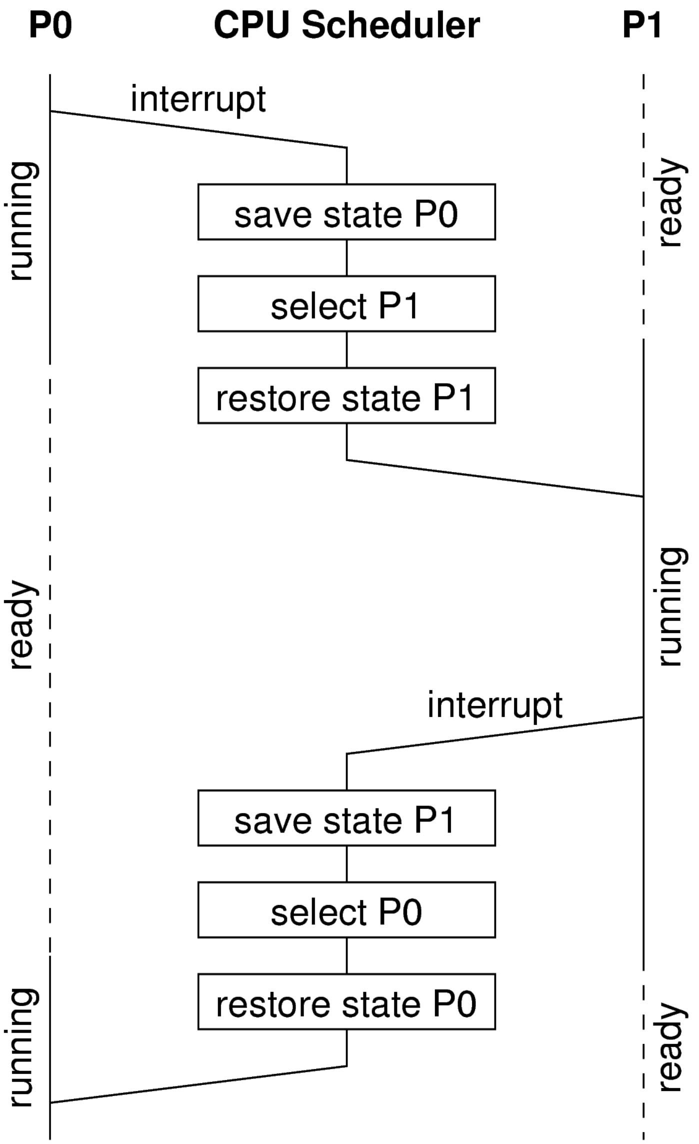


Process Context

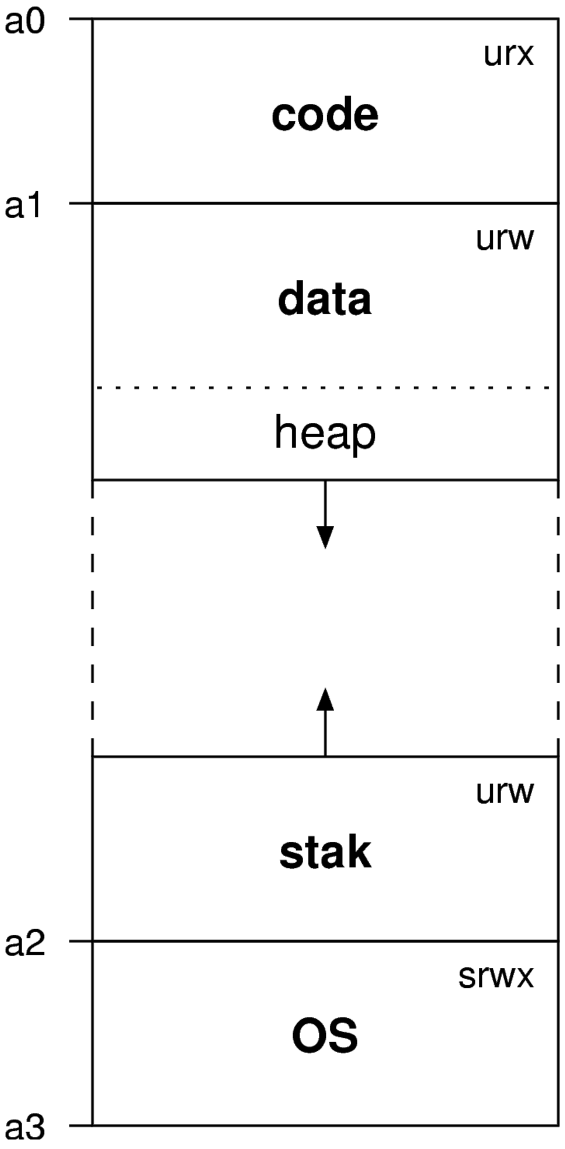
- Process context
  - Information that allows the OS to resume the execution of a process
  - Process Control Block (PCB)
    - State
    - CPU registers
    - Scheduling info
    - Memory info
    - I/O info
    - Accounting info
  - Process stack

Operating Systems

Context Switch



Process Address Space



← program instructions

← program global data

← program dynamically allocated data

← program local data

Process Creation

- A process is created when another process invokes the corresponding syscall (*e.g. fork*)
  - Creator = *parent* process
  - Created = *child* process
  - Child resources can be
    - Inherited from parent
    - Allocated from OS
  - Who creates the first process?
    - Forged by OS initialization procedure
- Process destruction
  - Natural: when a process terminates and calls *exit*
  - Forced
    - By the OS when a process misbehaves (*abort*)
    - By another process (parent) on convenience (*kill*)

Concurrent Processes

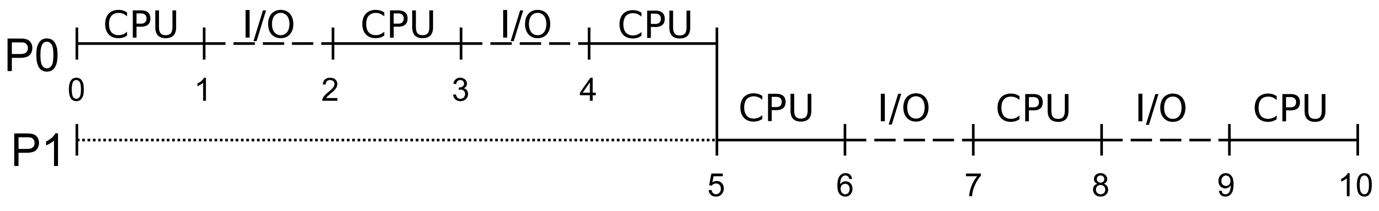
- Concurrent processes
  - Resource sharing (concurrence)
  - Speedup with multiple processing elements
- Independent process
  - A sequential program under execution
  - Private context
  - Output depends exclusively from input
- Cooperating processes
  - A parallel program under execution
  - Shared context
  - Output depends also on the relative execution order

Threads

- Threads
  - Also called lightweight process
    - Low creation overhead
  - Execution flow on a task
  - Share task's code, data and resources
  - Has its own stack
  - Traditional process = task + 1 thread

Multiprogramming

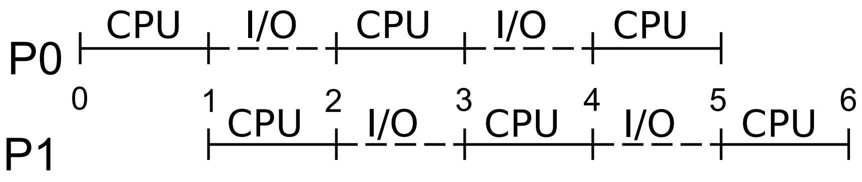
- Without multiprogramming



time = 10 tu
throughput = 0.2 proc/tu

average execution time = 7.5 tu
CPU usage = 50%

- With multiprogramming

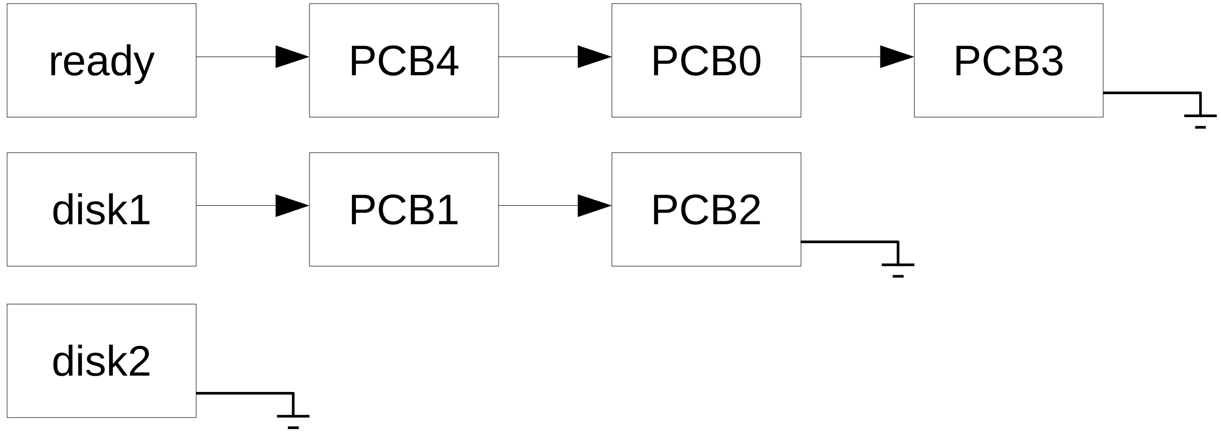


time = 6 tu
throughput = 0.33 proc/tu

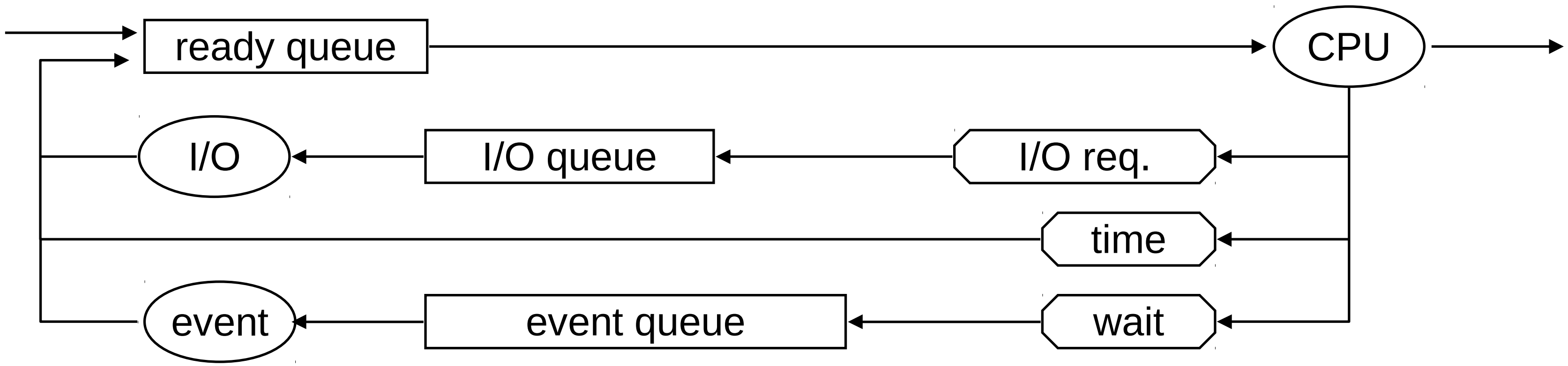
average execution time = 5.5 tu
CPU usage = 100%

Process Scheduling Structures

- Ready and I/O queues



- System queue diagram

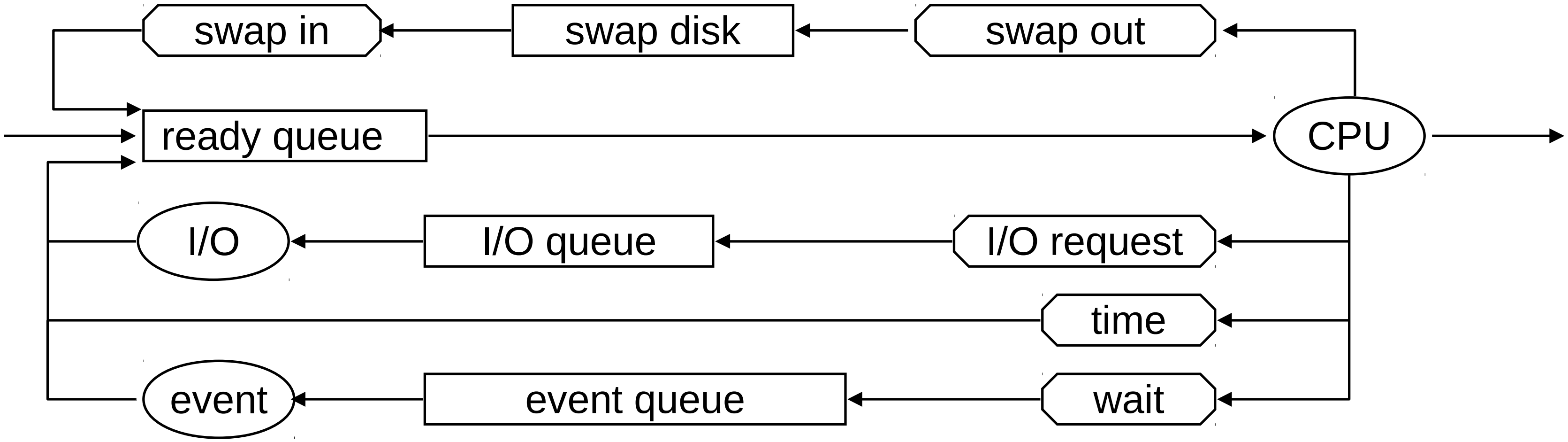


Process Schedulers

- Short term (CPU)
  - Selects processes from the ready queue
  - Runs very often and therefore must be very efficient
- Long term (jobs)
  - Selects processes that will be allowed in the system
  - Tries to balance I/O-bound e CPU-bound processes

Process Schedulers

- Medium-term (*swapper*)
  - Temporarily suspends processes
    - To keep the balance between I/O e CPU usage
    - Due to memory depletion



Preemptive and Non-preemptive Process Scheduling

- A process must be chosen to occupy the CPU whenever a process

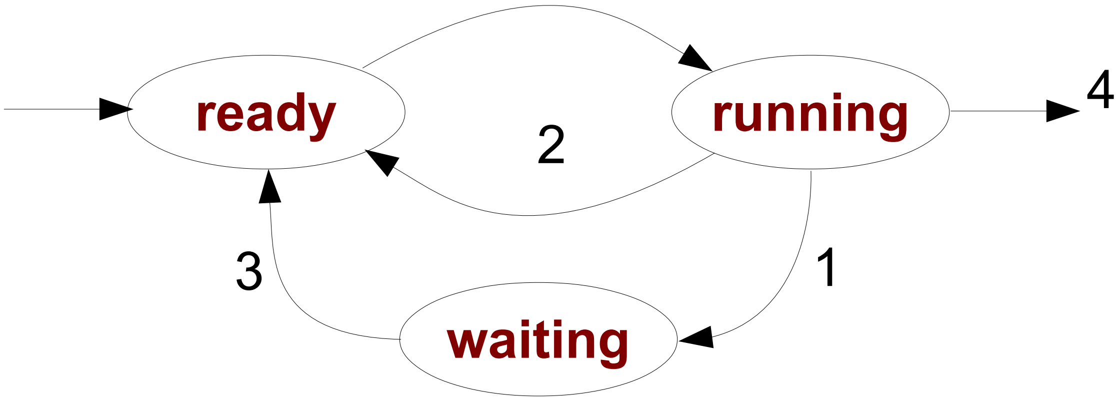
  1 - Changes state from *running* to *waiting*

  2 - Changes state from *running* to *ready*

  3 - Changes state from *waiting* to *ready*

  4 - Finishes

  - Preemptive scheduling: 1, 2, 3 and 4
  - Non-preemptive scheduling: 1 and 4



Process Scheduling Criteria

- Maximize CPU utilization
- Maximize system throughput (jobs/time)
- Minimize turnaround time (total time)
- Minimize waiting time (time waiting to run)
- Minimize and stabilize (user) response time

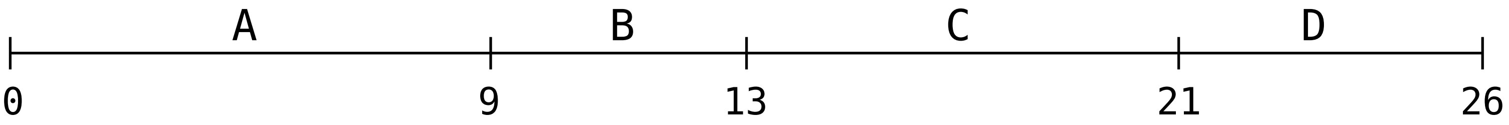
Process Scheduling Policy

- First Come First Served
- Shortest Job First
- Static Priority
- Dynamic Priority
- Round-Robin
- Multilevel Queue
- And thousands of derivations thereof

First Come First Served (FCFS)

- Policy
  - Ready queue under FIFO policy
  - New processes are inserted at the end
  - Non-preemptive
- Performance
  - Extremely poor when a CPU-bound process blocks an I/O-bound process
- Example

Process	A	B	C	D
CPU time	9	4	8	5
Arrival time	0	0	0	0

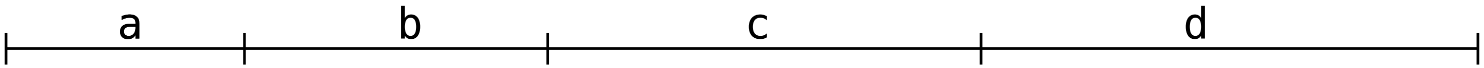


TA = (9 + 13 + 21 + 26) / 4 = 17.25 tu

WT = (0 + 9 + 13 + 21) / 4 = 10.75 tu

Shortest Job First (SJF)

- Policy
  - Process that will need the shortest CPU time is scheduled first
  - Preemptive or non-preemptive
- Performance
  - Optimal algorithm in terms of TA and WT



$$TA = (a + (a+b) + (a+b+c) + (a+b+c+d)) / 4 = (4a + 3b + 2c + d) / 4 \text{ tu}$$

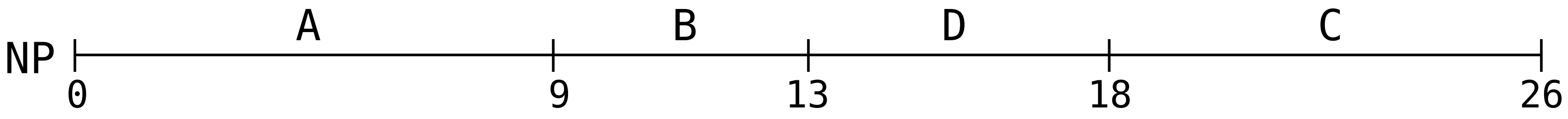
$$WT = (0 + a + (a+b) + (a+b+c)) / 4 = (3a + 2b + c) / 4 \text{ tu}$$

- Useful for processes for which the maximum execution time is known

Shortest Job First (SJF)

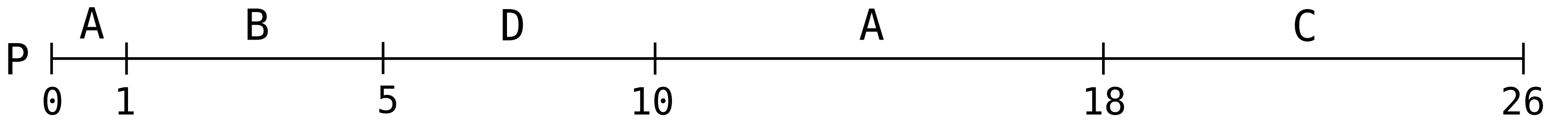
- Example

Process	A	B	C	D
CPU time	9	4	8	5
Arrival time	0	1	2	3



TA = (9 + 12 + 24 + 15) / 4 = 15 tu

WT = (0 + 8 + 16 + 10) / 4 = 8.5 tu



TA = (18 + 4 + 24 + 7) / 4 = 13,25 tu

WT = (9 + 0 + 16 + 2) / 4 = 6.75 tu

SJF Approximation

- Policy
  - Future estimation based on recent past
  - Process that has been having the shortest CPU cycles is scheduled first
- Formula

$$\pi_{n+1} = \alpha t_n + (1 - \alpha) \pi_n$$

π_{n+1} = next cycle estimate

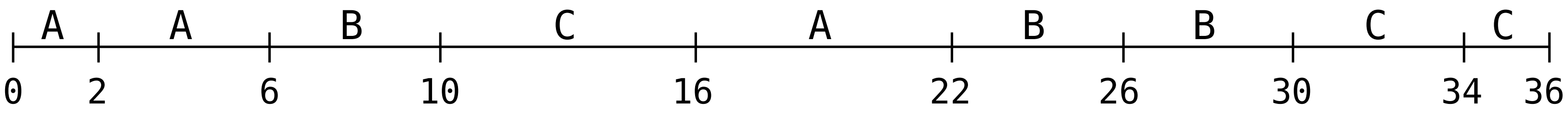
α = past importance factor

t_n = cycle *n* effective time

- Example (α = 1/2)

```
TA = (22 + 30 + 36) / 3 = 29.3 tu
WT= (10 + 18 + 24) / 3 = 17.3 tu
```

Process	π_0	t_0	π_1	t_1	π_2	t_2	π_3
A	1	2	1	4	2	6	4
B	1	4	2	4	3	4	3
C	1	6	3	4	3	2	2



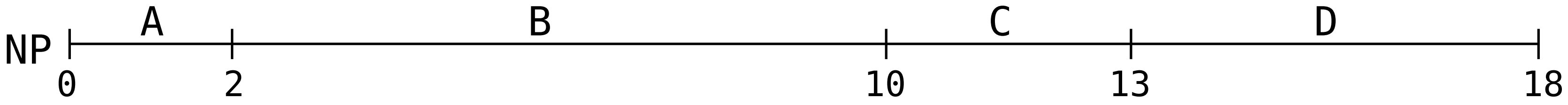
Priority

- Policy
  - Process with highest priority is scheduled first
  - Priorities can be assigned to processes either statically or dynamically
  - Preemptive or non-preemptive
- Processes might wait indefinitely
  - Low-priority processes only run when high-priority processes are *waiting*
- Typical of real-time systems

Static Priority

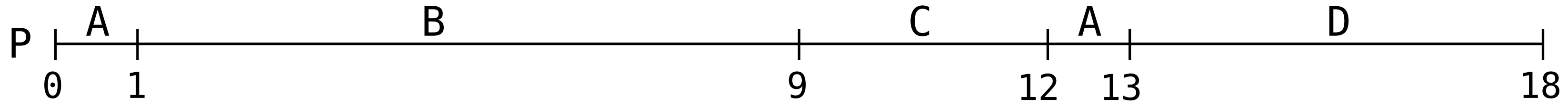
- Example

Process	A	B	C	D
CPU time	2	8	3	5
Priority	3	1	2	3
Arrival time	0	1	2	3



TA = (2 + 9 +11 + 15) / 4 = 9.25 tu

WT = (0 + 1 + 8 + 10) / 4 = 4.75 tu



TA = (13 + 8 + 10 + 15) / 4 = 11.5 tu

WT = (11 + 0 + 7 + 10) / 4 = 7 tu

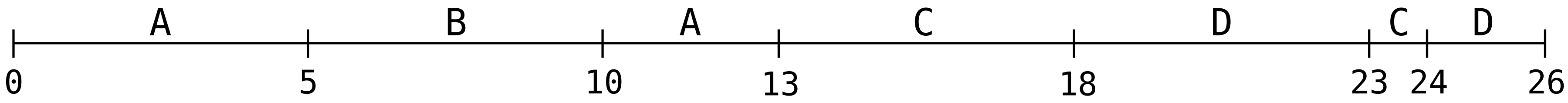
Round-Robin

- Policy
  - Processes are rescheduled periodically based on a time *quantum*
  - FIFO circular queue
  - Preemptive
- Formula
  - For a given set of processes with n elements and a time quantum of q:
    - Each process gets $1/n$ of CPU time in cycles that are no longer than q time units
    - Maximum waiting time = $(n - 1)\ q$
- Typical of time-sharing systems

Round-Robin

- Example (*q* = 5 tu)

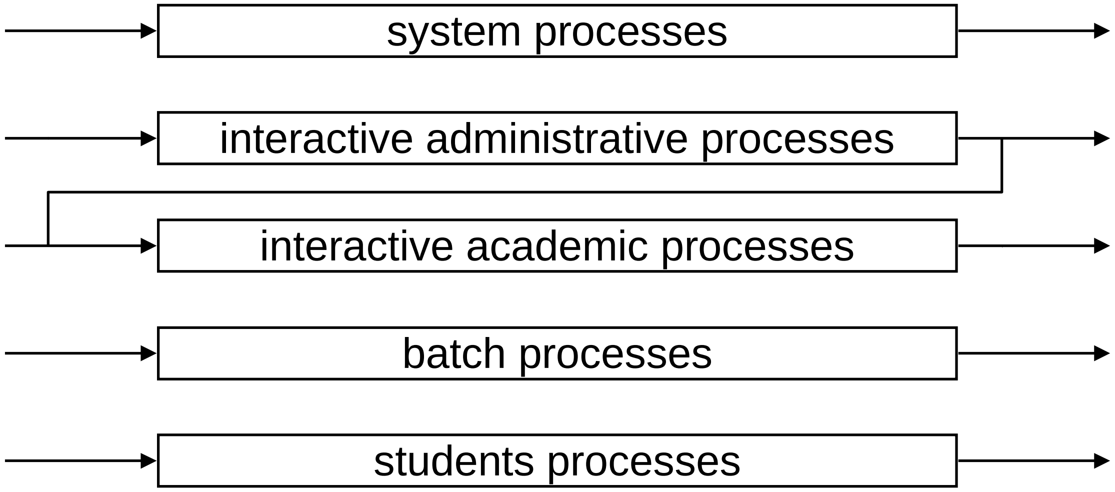
Process	A	B	C	D
CPU time	8	5	6	7
Arrival time	0	4	9	14



```
TA = (13 + 6 + 15 + 12) / 4 = 11.5 tu
Wt = (5 + 1 + 9 + 5) / 4 = 5 tu
```

Multilevel Queue

- Policy
  - Processes are grouped
    - E.g. system, interactive, batch
  - Each group has its own queue under a specific policy
  - Processes might be allowed to change groups


system processes
interactive administrative processes
interactive academic processes
batch processes
students processes

Process Synchronization

- Concurrent programs are executed by multiple cooperating processes that share some data
- Concurrent access to shared data may result in data inconsistency
- OS must provide mechanisms to synchronize and coordinate cooperating processes

Producer X Consumer

```
Producer:

shared int counter;
shared char buf[N];

int main()
{
  const int n = N;
  int in = 0;

  while (1) {
    while (counter == n);
    buf[in] = produce();
    in = ++in % n;
    counter++;
  }
}
```

```
Consumer:

shared int counter;
shared char buf[N];

int main()
{
  const int n = N;
  int out = 0;

  while (1) {
    while (counter == 0);
    consume (buf[out]);
    out = ++out % n;
    counter--;
  }
}
```

Race Conditions

```
Producer:                         Consumer:
  counter++;                      counter--
  load  R1,[counter]              load  R2,[counter]
  inc   R1                        dec   R2
  store R1,[counter]              store R2,[counter]


                                       R1 R2 [counter]
0) P: load  R1,[counter]               5  -  5
1) P: inc   R1                         6  -  5
2) C: load  R2,[counter]               6  5  5
3) C: dec   R2                         6  4  5
4) C: store R2,[counter]               6  4  4
5) P: store R1,[counter]               6  4  6
```

Critical Sections

- Sections of concurrent programs in which shared data is manipulated
- Conditions for proper execution
  - Mutual Exclusion: only a single process executes a critical section on a time
  - Execution progress: a process that is not executing a critical section cannot prevent others from doing it
  - Bounded waiting: a process cannot be deprived from execution a critical section indefinitely

Synchronization Algorithm I

Process 0

```
shared int turn;

int main()
{
  while (1) {
    while(turn != 0);

    /* critical */

    turn = 1;

    /* remainder */
  }
}
```

Process 1

```
shared int turn;

int main()
{
  while (1) {
    while(turn != 1);

    /* critical */

    turn = 0;

    /* remainder */
  }
}
```

- Misses the progress condition

Synchronization Algorithm II

```
Process 0

shared int flag[2];

int main()
{
  while (1) {
    flag[0] = 1;
    while(flag[1]);

    /* critical */

    flag[0] = 0;

    /* remainder */
  }
}
```

```
Process 1

shared int flag[2];

int main()
{
  while (1) {
    flag[1] = 1;
    while(flag[0]);

    /* critical */

    flag[1] = 0;

    /* remainder */
  }
}
```

- Misses the bounded waiting condition

Synchronization Algorithm III (Peterson)

```
Process 0

shared int turn;
shared int flag[2];

int main()
{
  while (1) {
    flag[0] = 1;
    turn = 1;
    while(flag[1] &&
                  turn);
    /* critical */

    flag[0] = 0;
    /* remainder */
  }
}
```

```
Process 1

shared int turn;
shared int flag[2];

int main()
{
  while (1) {
    flag[1] = 1;
    turn = 0;
    while(flag[0] &&
                  !turn);
    /* critical */

    flag[1] = 0;
    /* remainder */
  }
}
```

Synchronization Hardware

- Test and Set Lock (TSL) instruction

```
int tsl(int * ptr)
{
  int tmp = *ptr;
  *ptr = 1;
  return tmp;
}
```

- Usage

```
shared int lock = 0;
int main()
{
   while (1) {
       while(tsl(lock));
       /* critical */
       lock = 0;
   }
 }
```

Semaphores

- Integer variable accessible through atomic operations P and V

```
p(s): while(s <= 0);          v(s): s++;
      s--;
```

- Usage

```
shared int mutex;
int main()
{
    while(1) {
        p(mutex);
        /* critical */
        v(mutex);
        /* remainder */
    }
}
```

Semaphore Implementation

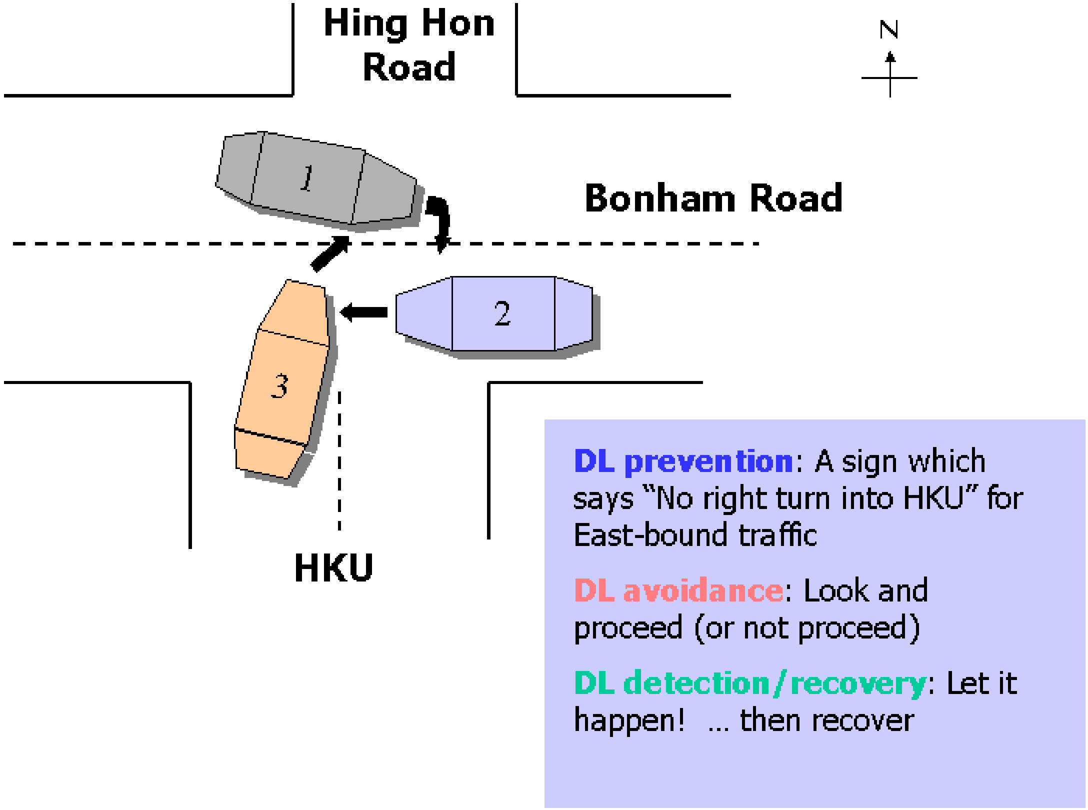
```
class Semaphore
{
public:
  Semaphore(int i) : s(i) {}
  void p();
  void v();
private:
  int s;
  list<Process> l;
};
extern Process * running;
```

```
void Semaphore::v()
{
  if(++s <= 0)
    l.pop()->wakeup();
}
```

```
void Semaphore::p()
{
  if (--s < 0) {
    l.push(running);
    running->sleep();
  }
}
```

Deadlocks

- A deadlock occurs when two or more processes are waiting for an event that can only be generated by one of the waiting processes



DL prevention: A sign which says "No right turn into HKU" for East-bound traffic

DL avoidance: Look and proceed (or not proceed)

DL detection/recovery: Let it happen! ... then recover

Deadlock Characterization

- Resource allocation
  - Request => Use => Release
- Conditions
  - Mutual exclusion: resources cannot be shared
  - Hold and wait: a process holds some resources but needs a resource that is held by another process
  - No preemption: resources cannot be preempted
  - Circular wait: there must be a circular chain of processes, each of which is waiting for a resource held by the next in the chain

Deadlock Handling

- Prevention
  - Ensure that at least one of the conditions necessary to characterize a deadlock will never hold
- Detection and recovery
  - Allows deadlocks to occur
  - Detection algorithm is run periodically
    - Allocated resources X waiting processes
  - Recovery algorithm is run whenever a deadlock is detected
    - Process termination
    - Resource preemption (rollback)
- Practice
  - Too expensive, seldom used!

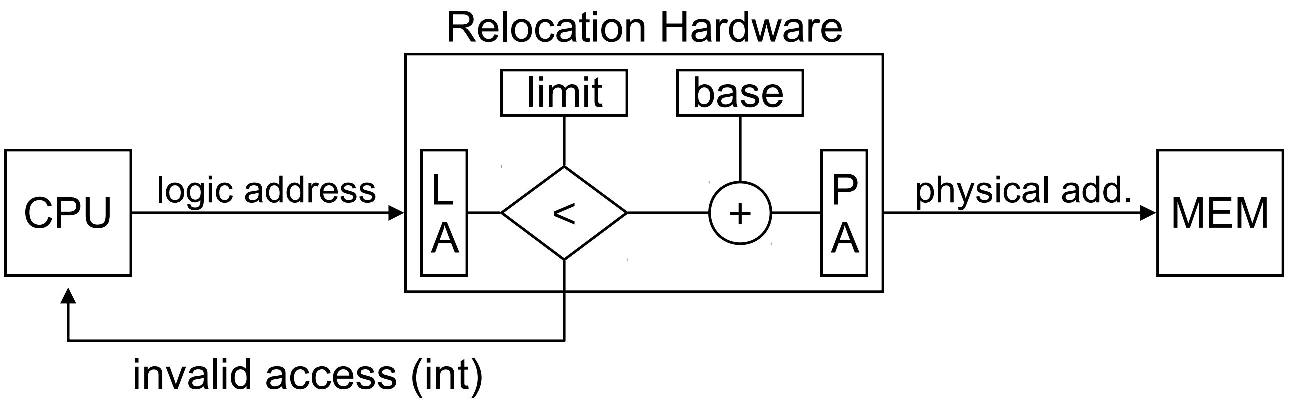
Memory Management

- Processor fetches instructions from main memory
  - Programs must be loaded into memory before they can be executed
- Address referenced by programs (e.g. variables) must be bound to memory
  - Compile-time: absolute addresses
  - Load-time: relocatable code
  - Run-time: relocation hardware
- Programs bigger than memory
  - Overlays are replaced during program execution
  - Might be supported by the OS
- Often replicated programs
  - Can be organized as shared libraries

Operating Systems

Single-Process Memory Allocation

- Without OS support
  - Simple dedicated systems (e.g. embedded)
  - No memory manager
- With OS support
  - OS memory is protected through a base register
  - User process is loaded just after OS
- Dynamic relocation with hardware support
  - Compiler and CPU issue *logic* memory addresses
  - Relocation hardware adds logic address to a base address to generate *physical* memory addresses



Multi-Process Memory Allocation without Hardware Support

- List of free memory blocks
  - First-fit: allocates the first block that is large enough to hold the process
  - Best-fit: allocates the smallest block that is large enough to hold the process
  - Worst-fit: allocates the largest available block
- External fragmentation
  - Large amount of small-size blocks
  - Enough free memory to satisfy a request but not contiguously
  - Tend to 1/3 of all the memory for n-fit algorithms

Multi-Process Memory Allocation without Hardware Support

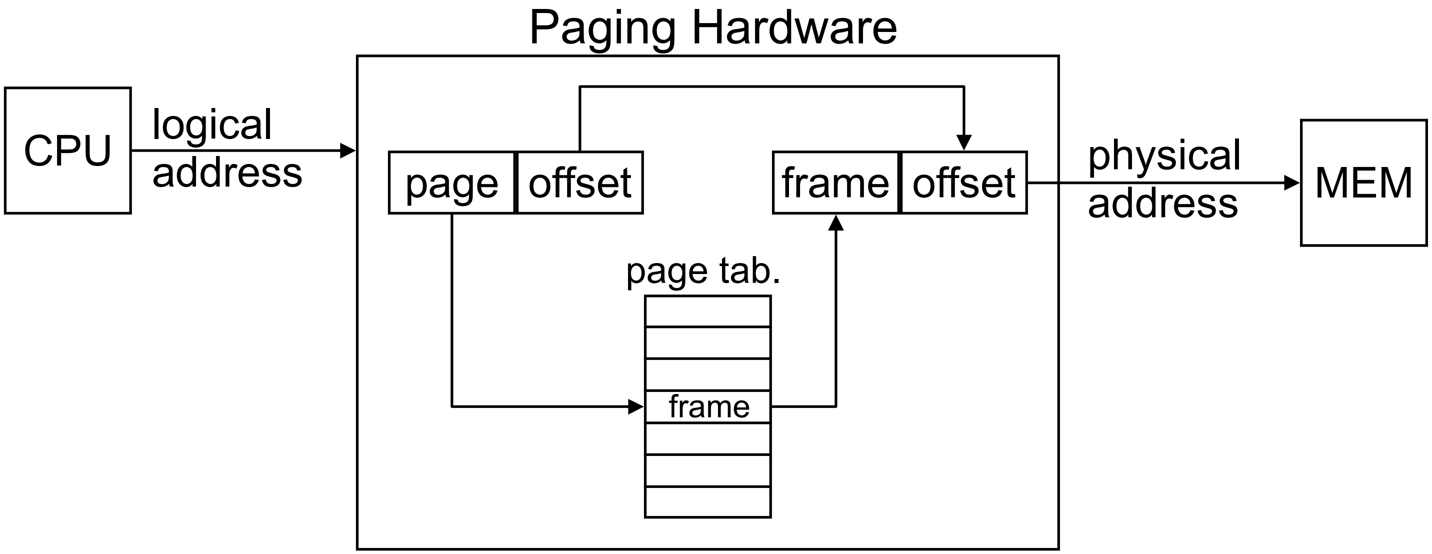
- Protection
  - Through base and limit registers
- Compaction
  - Dynamically relocates processes in order to group free blocks together
  - Relocation support
    - Relative addresses-only (implicitly relocatable)
    - Re-linking by OS application loader (expensive)
    - External relocation support

Multi-Process Memory Allocation with Hardware Support

- Detachment of address space and memory concepts
  - Compilers, processors and processes operate on an address space that is mapped into memory by an MMU
- Memory Management Unit (MMU)
  - Translates logical addresses into physical ones
  - Thus maps processes' address spaces into memory
- Typical strategies
  - Paging
  - Segmentation
  - Paged segmentation

Paging

- Memory organized and allocated in pages
- Processes address spaces
  - Organized in pages
  - Mapped into frames (physical memory pages) through page tables



Paging and Fragmentation

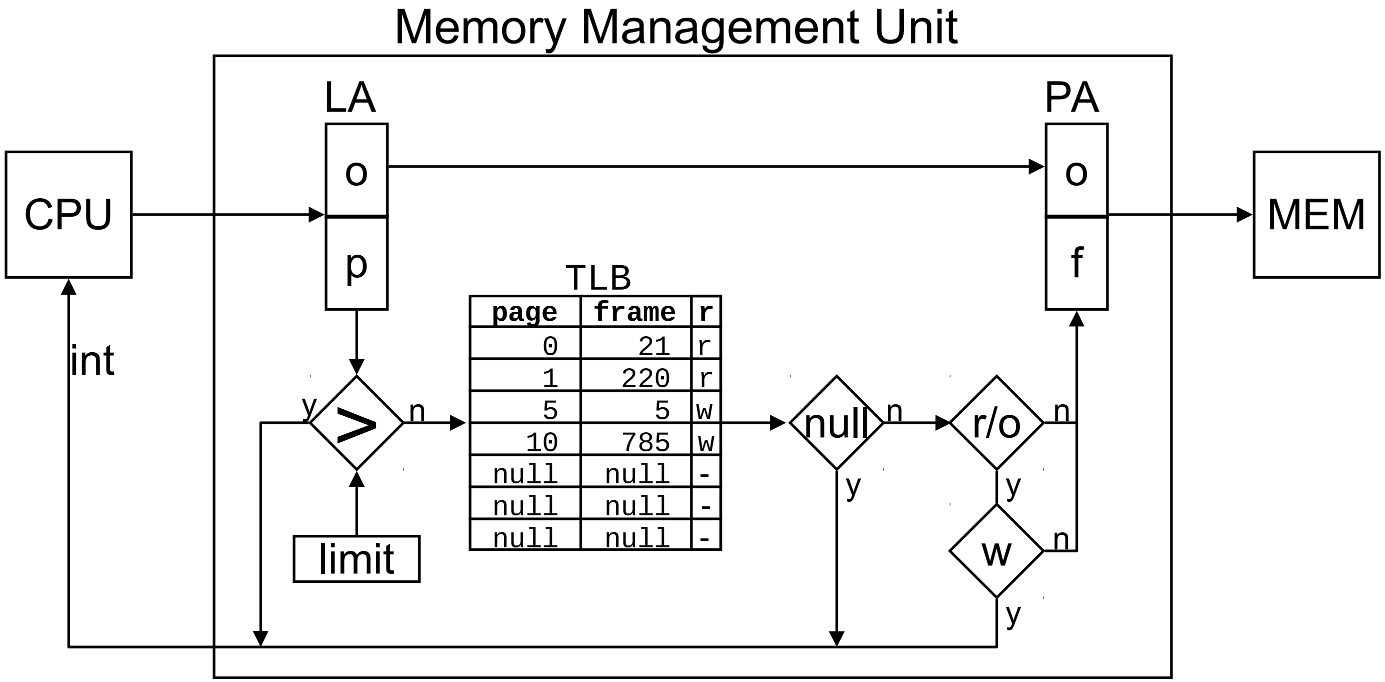
- No external fragmentation
- Internal fragmentation
  - Unused fraction of a page that cannot be allocated to other processes
- Internal fragmentation x page size
  - Small page size -> less fragmentation -> more memory for page tables
  - Example
    - Allocation request for 1 Gbyte
    - Pages of 4 Kbytes -> 256 Kpages
    - Pages of 4 Mbytes -> 256

Paging Implementation

- Page tables
  - Registers: limited to few pages (small address spaces or large pages)
  - Memory: slow (double memory access time)
  - Translation Look-aside Buffer (TLB)
    - Cache of page translations
    - Fast and expensive (fully associative memory)
    - Good performance if hit ratio is high (replacement policy)
- Page sharing
  - Page tables of distinct processes can reference common pages
  - Explored by shared libraries for immutable code
- Protection
  - Pages are tagged with permission bits that are checked by the MMU
  - Limit register to reduce page table size

Paging Example

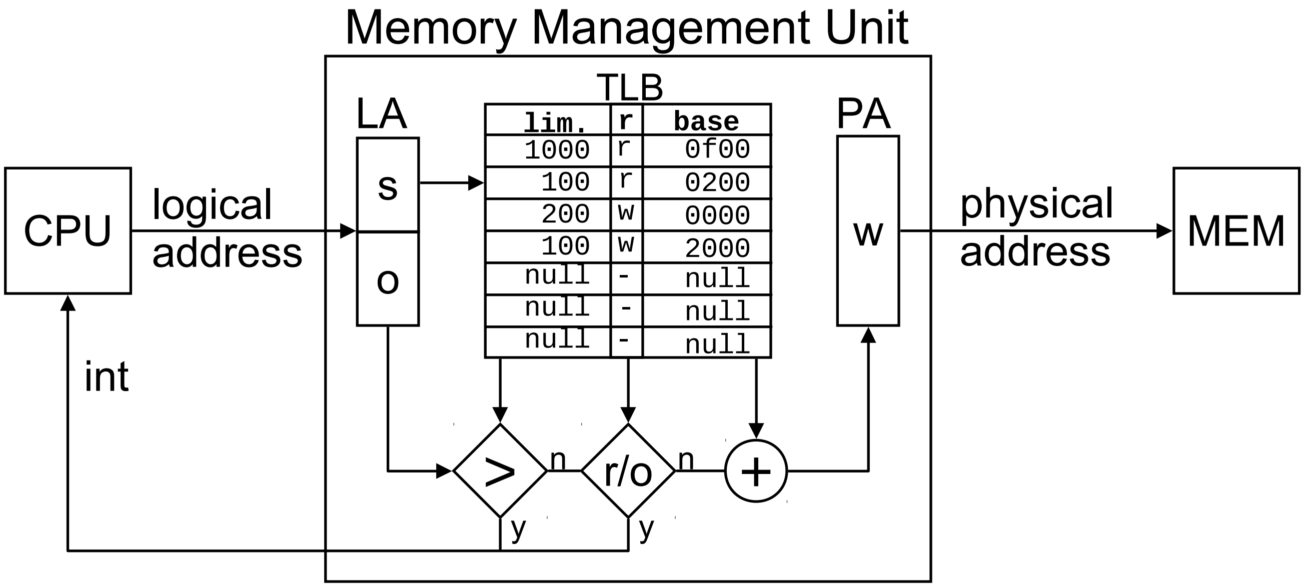
Frame size: 4 Kbytes
Memory size: 4 Gbytes (1 Mframes)
Address space size: 64 Mbytes (16 Kpages)
Physical address (PA): 32 bits (frame = 20, offset = 12)
Logical address (LA): 26 bits (page = 14, offset = 12)
Page table size: 16 Kentries



page	frame	r
0	21	r
1	220	r
5	5	w
10	785	w
null	null	-
null	null	-
null	null	-

Segmentation

- Memory organized in segments and allocated in words
- Bi-dimensional addresses: (segment, offset)
- Processes address spaces
  - Organized in segments
  - Mapped into physical memory through segment tables with base and limit for each segment



lim.	r	base
1000	r	0f00
100	r	0200
200	w	0000
100	w	2000
null	-	null
null	-	null
null	-	null

Segmentation and Fragmentation

- External fragmentation
  - One segment per process -> n-fit (1/3)
  - Fixed-size segments -> paging
  - Word-size segments -> large segment tables and double memory access time
  - One segment per object
    - Intel proposal
    - Not implemented by ordinary compilers
- No internal fragmentation
  - Limit can be adjust to fit used fraction of segment

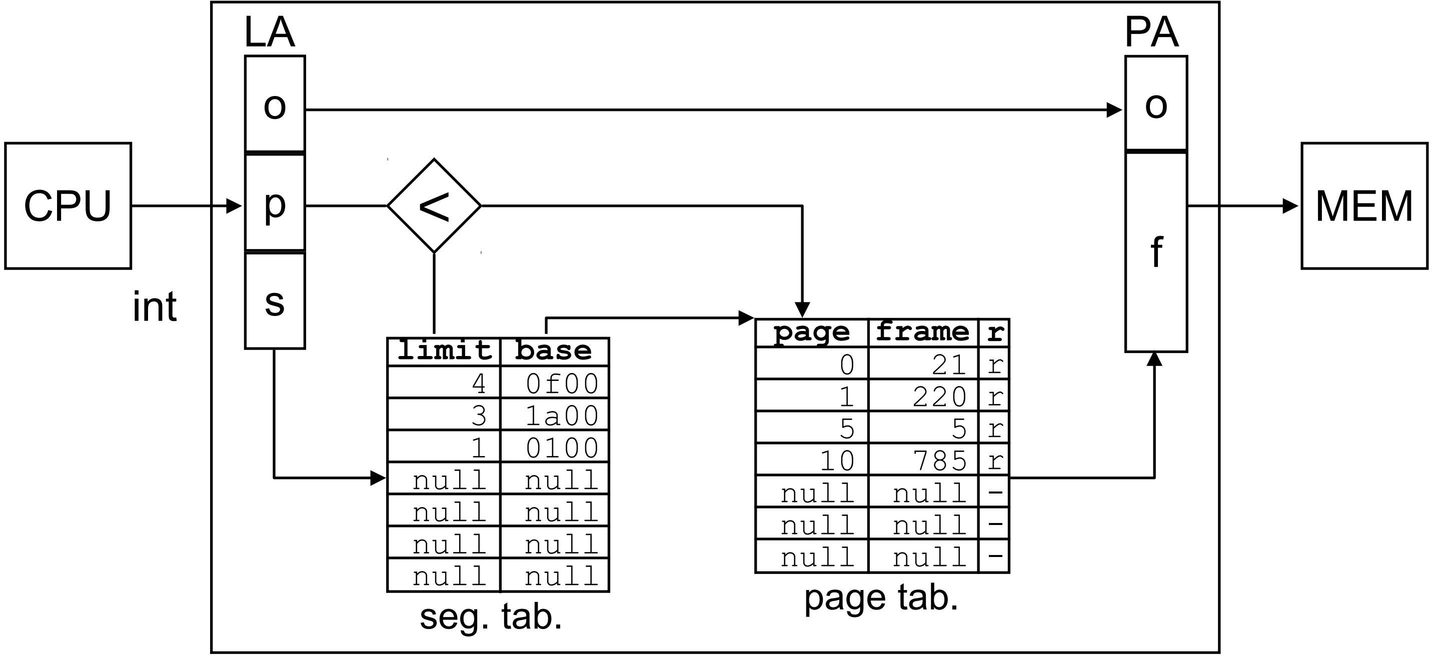
Segmentation Implementation

- Segment tables
  - Registers: limited to few segments (small address spaces or large segments)
  - Memory: slow (double memory access time)
  - Translation Look-aside Buffer (TLB)
    - Cache of address translations
    - Fast and expensive (fully associative memory)
    - Good performance if hit ratio is high (replacement policy)
- Segment sharing
  - Segment tables of distinct processes can reference common segments
- Protection
  - Segments are tagged with permission bits that are checked by the MMU
  - Segment limits are also checked by the MMU

Paged Segmentation

- Merger of segmentation and pagging
- Address spaces of processes are split in segments, each of which is subsequently paged
  - Segment tables point to page tables

Memory Management Unit



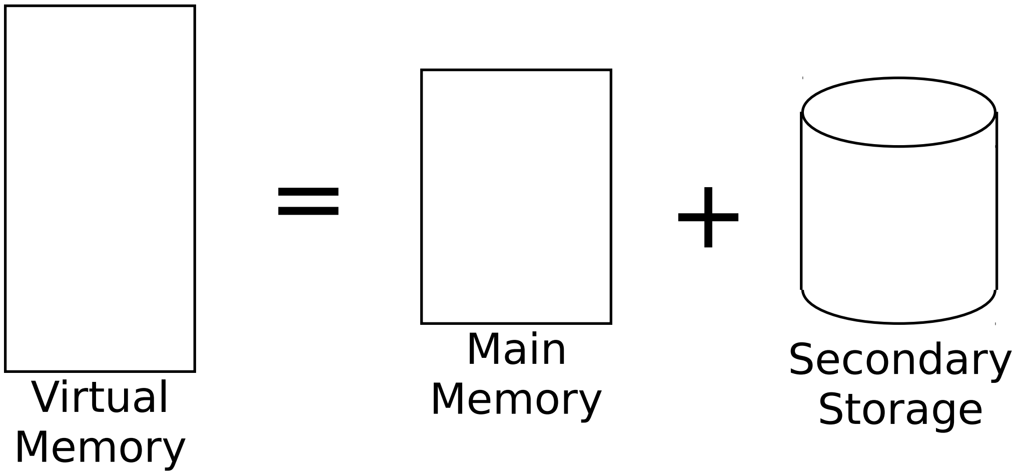
limit	base
4	0f00
3	1a00
1	0100
null	null
null	null
null	null
null	null

seg. tab.

page	frame	r
0	21	r
1	220	r
5	5	r
10	785	r
null	null	-
null	null	-
null	null	-

page tab.

Virtual Memory



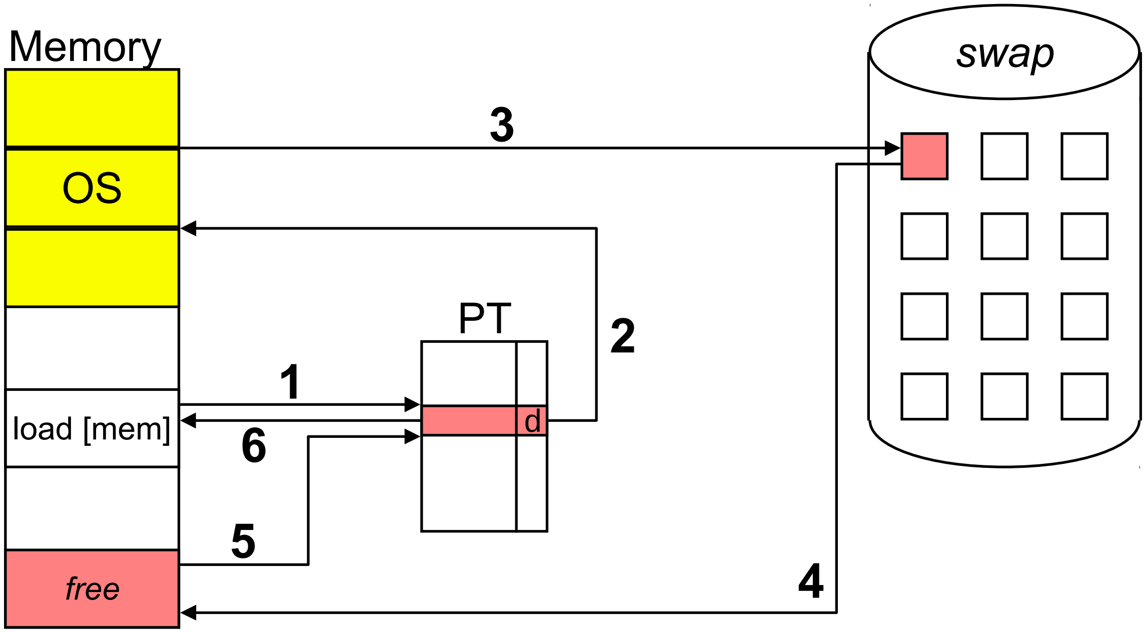
- Allows a process to be executed even if not completely loaded into memory
- Allows for processes to allocate more memory than the size of physical memory
- Can improve CPU utilization
- Can reduce swap overhead
- Can kill your system!

Swapping

- Processes can be temporarily suspended and the memory allocated to them is first copied to a secondary storage (i.e. disk) and than released to other processes (*swap out*)
  - Sleeping processes
  - Low-priority processes
- Such processes can be latter resumed by restoring their address spaces from the copy in the secondary storage (*swap in*)

Demand Paging

- Page-oriented swapping
  - Page table flag indicates whether the page is in memory or on disk
  - MMU triggers an exception (*page-fault*) whenever an absent page is accessed



1 – reference
2 – page-fault
3 – I/O request
4 – page in
5 – page table update
6 – instruction resume

Page-Fault Handling

Page-fautl trap	1µs
Save context	10µs
Dispatch PF handler	1µs
Locate page on disk	50µs
Read page from disk	10ms
Waiting queue	0s
Seek	7ms
Latency	2ms
Transfer	1ms
Scheduler	15µs
Disk I/O completion	1µs
Save context	10µs
Dispatch disk I/O handler	1µs
Update page table	15µs
Ready queue waiting	0s
Scheduler	15µs
TOTAL	10,109ms

Demand Paging Performance

- Formula

$$\texttt{eat = (1 - p) x mat + p x pft}$$

eat = effective access time
mat = memory access time
pft = page-fault handling time
p = page-fault probability

- Example

mat = 50 ns
pft = 10 ms = 10.000.000 ns
eat = (1 - p) x 50 + p x 10.000.000
eat = 50 + 9.999.950 x p
p = 0.001 -> eat = 10 ˪ s
eat = 50 ns -> p < 0,000.005 (1 / 200.000)

Page Replacement

- Whenever necessary, OS selects pages to be moved out to disk and than make them available to processes
- Algorithm criteria
  - Minimize page-fault ratio
  - Minimize I/O
- Dirty bit
  - Each page status (modified or not) is kept in the corresponding entry in the page table
  - MMU automatically sets that bit whenever a page is modified (written)
  - Modified (dirty) pages must be written to disk before being reused

First-In First-Out (FIFO)

- Replace the page that has been longer in memory (e.g. pages are time-stamped)
- Implemented using a FIFO queue
- Example

Accesses **15 *pf***

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2	2	4	4	4	0			0	0			7	7	7
	0	0	0		3	3	3	2	2	2			1	1			1	0	0
		1	1		1	0	0	0	3	3			3	2			2	2	1

- Increasing memory may not decrease pf rate

Aaccesses **9 *pf***

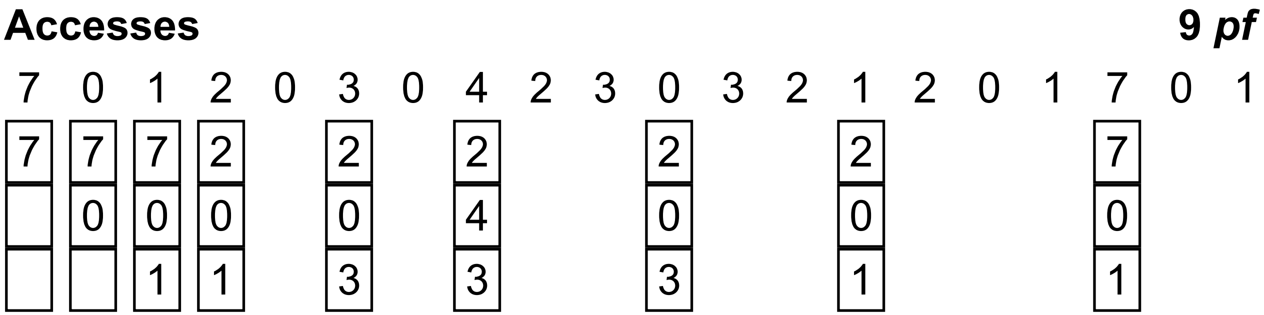
1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5			5	5	
	2	2	2	1	1	1			3	3	
		3	3	3	2	2			2	4	

Accesses **10 *pf***

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1			5	5	5	5	4	4
	2	2	2			2	1	1	1	1	5
		3	3			3	3	2	2	2	2
		4	4			4	4	4	3	3	3

Optimal

- Replace the page that will not be used for the longest period of time
- Not implementable, for it relies on knowing the future
- Example



Accesses **9 *pf***

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		2			2			2				7		
	0	0	0		0		4			0			0				0		
		1	1		3		3			3			1				1		

Least Recently Used (LRU)

- Uses the recent past as an approximation of the near future
- Replace the page that has not been used for the longest period of time
- Example

Accesses **12 *pf***

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2		2		4	4	4	0			1		1		1		
	0	0	0		0		0	0	3	3			3		0		0		
		1	1		3		3	2	2	2			2		2		7		

- Implementation
  - Time-stamp for each page
  - Linked-list of pages

LRU Approximations

- Reference bit
  - Each page is assigned a reference bit that is set by the MMU whenever the page is accessed
  - OS clears those bits periodically
  - Order of use is unknown
  - Target page is any with cleared reference bit
- Reference word
  - Additional reference bits that are shifted by the MMU
  - Target pages are those with smallest reference values
- Second chance
  - Pages are tracked by a circular FIFO list
  - If the pointed page has a clear reference bit, it is taken to be replaced
  - Otherwise, the bit is cleared and the pointer is adjusted to the next page

LRU Approximations

- Least Frequently Used (LFU)
  - Uses a reference counter for each page that is incremented by the MMU
  - Target page is the one with smallest counter value
  - Pages intensively accessed in the past, but no longer in use will take long to be replaced
- Reference and Modification bits
  - In addition to accesses, MMU marks pages that have been modified
  - Replacement order
    - Not-accessed, not-modified
    - Not-accessed, modified
    - Accessed, not-modified
    - Accessed, modified

Allocation of Frames

- Minimum set of frames
  - Instructions and operands may be scattered across several pages
  - Architecture-dependent
  - Instructions must be restarted after a page-fault
- Frames per process
  - Proportional to process size (i.e. memory footprint)
  - Equal to all processes
- Process interference
  - A process might cause the replacement of a page initially allocated to other process
  - A process may only replace its own pages

Thrashing

- A process is “thrashing” if it spends more time replacing pages the executing
- Causes
  - OS monitors CPU utilization and allows more processes in
  - If CPU utilization was low do to page-faults, increasing the number of processes in the system might cause thrashing
  - Thrashing only occurs if global page replacement (i.e. from other processes) is allowed
- Prevention
  - At any given time, a running process must have a set of pages available that fulfills its demands: its working set of pages (time x space locality)

Final Considerations

- Process load
  - On-demand
  - At-once to main memory
  - At-once to swap disk
- Page size
  - Large -> less page-fault, less I/O, less page tables
  - Small -> less internal fragmentation
- I/O results
  - Pages that will recieve I/O results must be pinned-down
- Programming and code generation
  - Although virtual memory is functionally transparent to programs, memory access patterns might have big influence on it (e.g. matrices)

File Management

- Motivation
  - Common interface to transparently manipulate data on secondary storage
- File system
  - Abstract a storage device (e.g. disk) as
    - A collection of files (data) plus
    - A directory structure (control information)
  - Interaction with storage devices through services exported by the corresponding device drivers
    - Device = linear array of blocks
  - One of the most visible OS structures
  - Examples:
    - FAT, UFS, EXT2, NTFS, ISO9660, etc

Files

- File
  - Named, nonvolatile sequence of bits, bytes, lines or records
- Typed file
  - Internal structure defined by the OS
    - Executable files, graphics files, text files, etc
  - Limited number of known types
- Untyped file
  - Streams of bytes whose meaning is defined by the user
  - Unlimited and flexible

File Attributes

- Name
  - Character string identifying the file to users
- Type (only for typed files)
  - OS internal type information
- Location on device
- Size
- Ownership
- Access control
  - Who can access the file for what operations
- Access history
  - Dates, times, users, counters, etc

File Operations

- Creating
  - Locate space in the file system
  - Create a directory entry
- Deleting
  - Search the directory for the named file
  - Release file system space
  - Remove the corresponding directory entry
- Writing/reading
  - Search the directory for the named file
  - Determine the location in the file system to operate
  - Write/read data
  - Update the file pointer

File Operations

- Positioning
  - Search the directory for the named file
  - Move the file pointer
- Opening/closing
  - Since all file operations require a directory search, it is usual to implement these operations to fetch significant file's information into the system 'table of open files'
- Memory mapping
  - Associate a portion of a process' address space with a section of a file, so that reading and writing to that memory region is equivalent to performing the corresponding operations on the file

File Access Methods

- Sequential
  - Ordered access, one record after the other (tape model)
  - File pointer incremented after each operation
  - Rewind moves the file pointer back to the beginning
- Direct
  - File pointer can be moved arbitrarily (disk model)
- Indexed
  - Based on the direct access method
  - Index associating a search key to records

File Consistency Semantics

- Unix
  - Writes to an open file by a user are immediately visible to all other users that have that file open
  - Locking mechanism for access synchronization
- Session (Andrew)
  - Every new open returns a 'copy' of the file
  - No file concurrency (private copy)
  - Update at close (visible to new sessions)
- Immutable-shared-file (Bullet)
  - Shared files are made read-only

File Access Control

- Motivation
  - Multiuser file system call for access control by the OS
- Types of access
  - Read, write, execute, append, delete
- Access criteria
  - Knowing the name of files
  - Knowing a password associated to files or directories
    - Impractical for interactive applications
  - Being included in a file or directory access list
    - Associate users and access permissions
    - Hard to maintain
    - Variable size structures

File Access Control

- Unix approach
  - Simplified access list
    - Permissions to reading, writing (deleting), and executing (entering)
    - Permissions for owner, owner's group, and others
  - Example

    `drwxr-xr-x dir1` owner can write, all can read and navigate

    `-rwxrwxrwx fil1` all can do everything

    `-r-x------ fil2` owner can execute

    `-r--r--r-- fil3` all can read

Directories

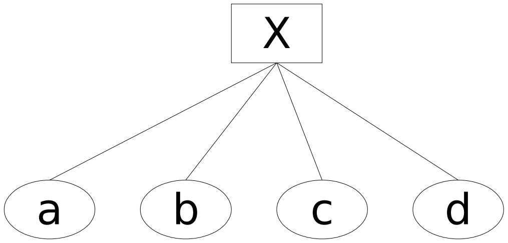
- Directory
  - Collection of information about files
  - Translation table (name => control info)
- Device directory
  - Files' physical characteristics
    - Size, location on disk, owner, etc
- File directory
  - Volume's table of contents
  - Associate file names with device directory entries

Directory Operations

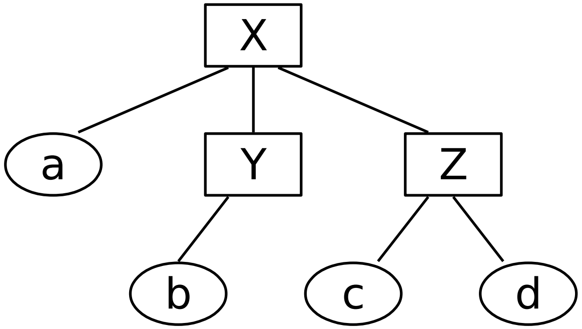
- Create/delete directories
- Add and remove directory entries
  - For file creation/deletion
- Manipulate directory entries
  - For file renaming or control information updating
- Search for a file or pattern
- Listing
- Traversing
  - For file system-wide operations such as search and backup

Directory Organization

- Flat
  - Single directory with all files

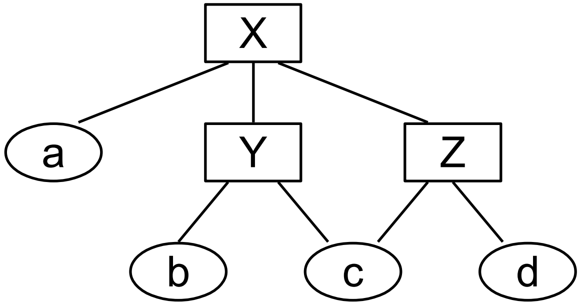


- Tree
  - OS differentiates nodes (directories) and leaves (files)
  - Root node ('/')
  - Pathnames
    - Absolute ('/')
    - Relative to current directory ('CWD')



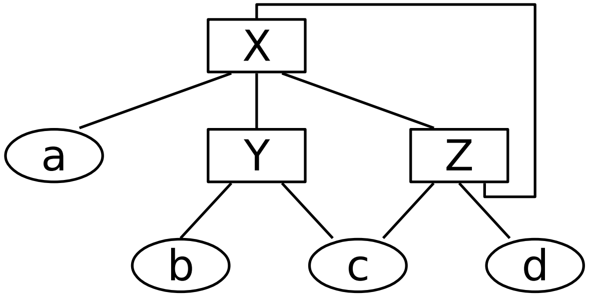
Directory Organization

- Acyclic graph
  - Hard link
    - Reference counter
    - File and link are indistinguishable
    - Not applicable to directories
  - Symbolic link
    - Pathname
    - File and link can be distinguished
    - May become broken
  - Name aliasing problems
    - Deleting
    - Traversing



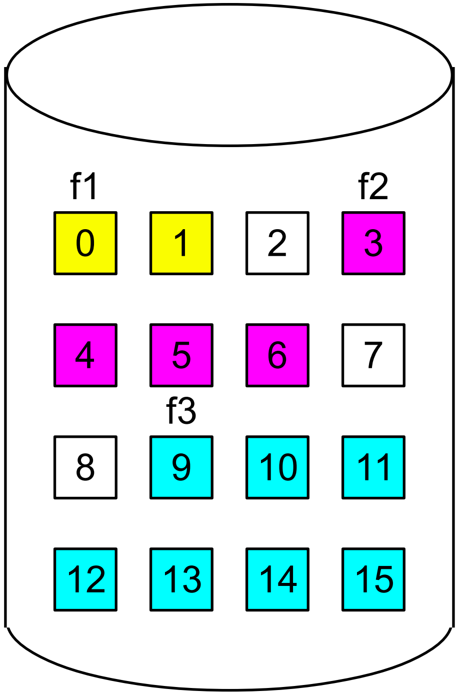
Directory Organization

- General graph
  - Cycles are allowed to exist in the directory
    - Hard links to directories
  - Search algorithm must detect cycles
    - Avoid infinite loops
  - Garbage collection (self reference)



Block Allocation Methods

- Contiguous allocation
  - Directory = (name, start, length)
  - Optimal sequential access plus direct access
  - File size defined at creation-time
    - A sufficiently large set of contiguous blocks must be located (first/best/worst-fit)
  - External fragmentation
    - Garbage collection

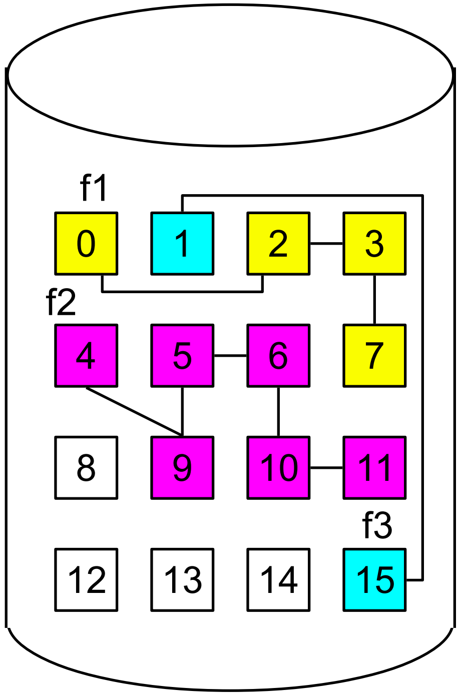


Directory

file	start	length
f1	0	2
f2	3	4
f3	9	7

Block Allocation Methods

- Linked allocation
  - Directory = (name, start, end)
  - Files are linked lists of blocks
    - Any block can be linked to any file
    - No external fragmentation
  - No direct access
  - Limited reliability

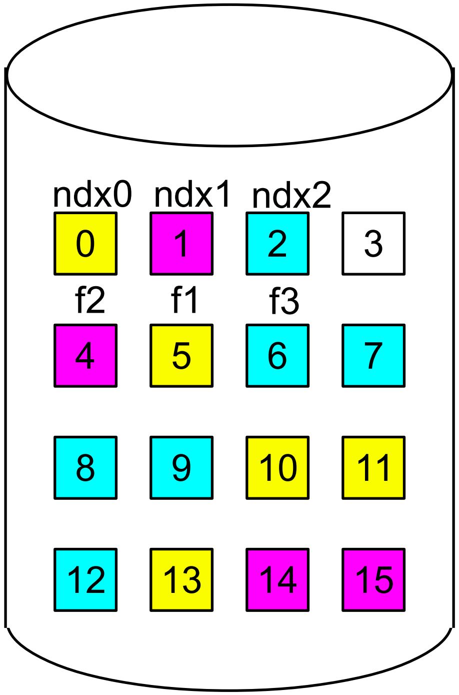


Directory

file	start	end
f1	0	7
f2	4	11
f3	15	1

Block Allocation Methods

- Indexed allocation
  - Directory = (name, index) + index
  - Similar to paging
  - Direct access without external fragmentation
  - Large files
    - Linked index
    - Multilevel index



Directory

file	index
f1	0
f2	1
f3	2

index0
5
10
13
11
-

index1
4
14
15
-
-

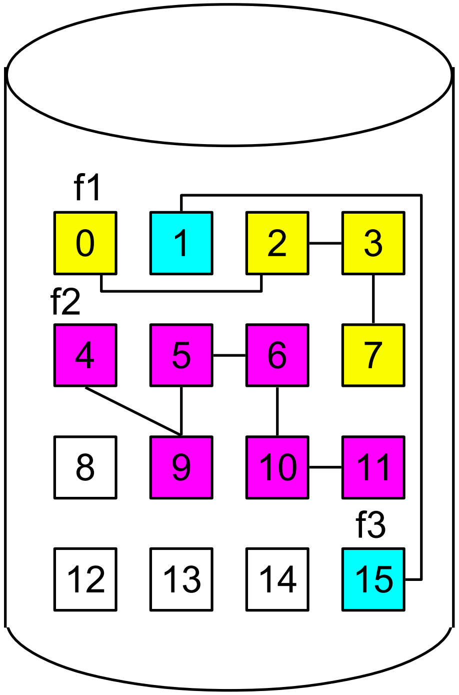
index2
6
7
8
9
12

Free-Block Management

- Bit map
  - Each block is represented by one bit (free/used)
  - Easy to locate sequences of same-state bits
    - Supports contiguous allocations
    - Optimizes sequential access
  - Must reside in memory to be efficient
- Linked list
  - Free blocks are linked in a list
  - Allocation and releasing imply in I/O
- Grouping
  - First free block groups a set of free blocks and contains a pointer to the next grouping block
  - Contiguous ranges of blocks can be represented as pointer + count

Case Study:
MS-DOS File Allocation Table (FAT)

- MS-DOS FAT
  - Directory =
    (name, start, end) + FAT
  - Table with on entry per block is kept separately
  - Special values for free blocks and end of files
  - Allows direct access
  - Reliability improved by replication



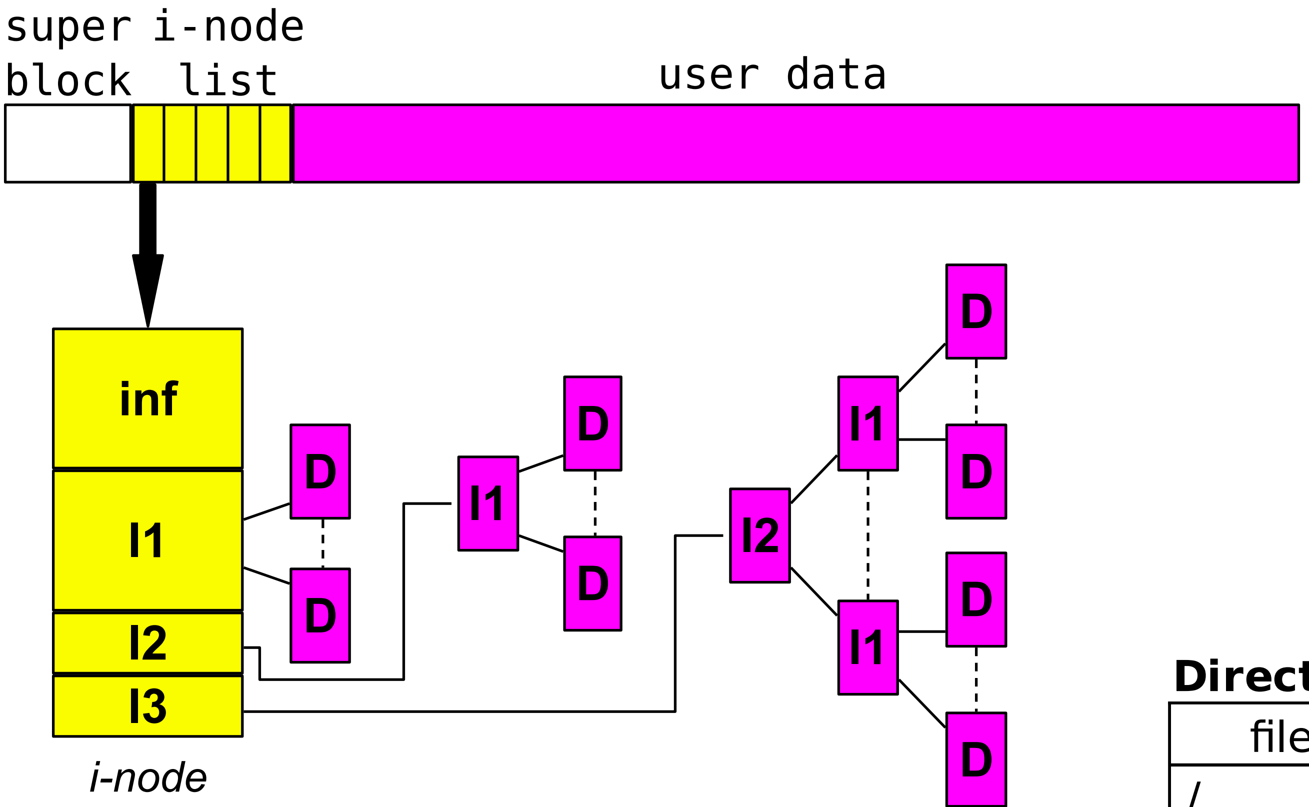
Directory

file	start	end
f1	0	7
f2	4	11
f3	15	1

FAT

2	eof	3	7
9	6	10	eof
free	5	11	eof
free	free	free	1

Case Study: Unix File System



Directory

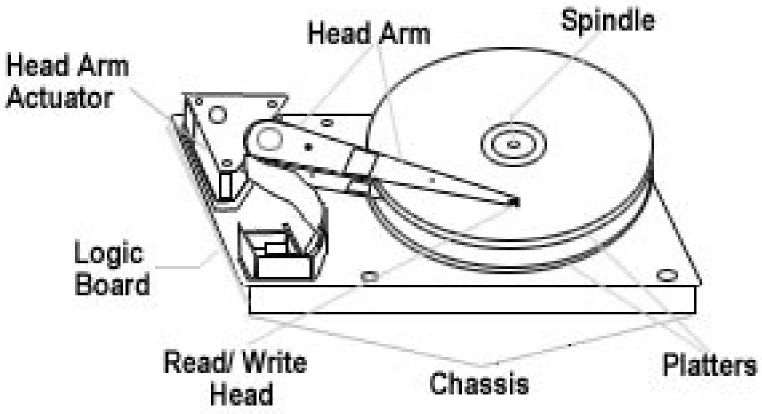
file	i-node
/	0
f1	1
f2	2

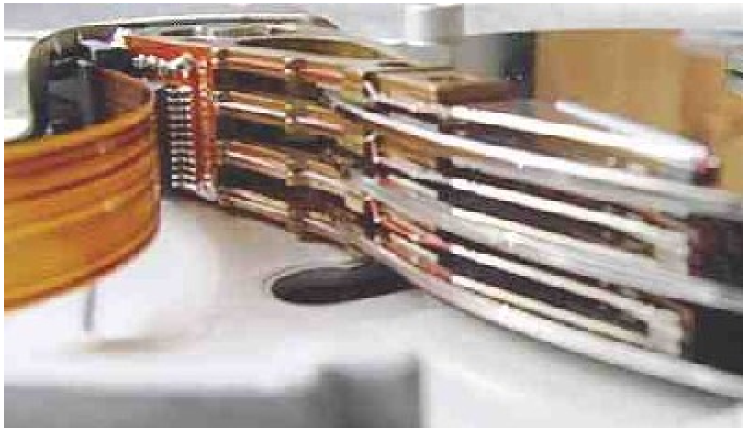
Secondary Memory Management

- Motivation
  - Main memory is small (expensive) and volatile
  - Secondary memory is large (cheap) and persistent
    - Typically disks
- Operational basis for important OS components
  - Swapping
  - Virtual memory
  - File system

Disk Drives

- Physical structure
  - Media
    - hard or flexible, fixed or removable
  - Driver (mechanical)
    - disk rotation and head positioning
  - Controller (electronic)
    - operation and host interfacing
- Technologies
  - Magnetic
  - Optic
  - Optomagnetic

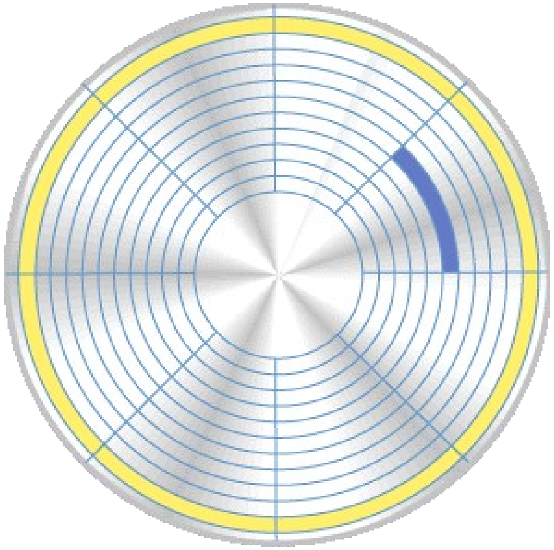




Disks

- Physical structure
  - Concentric tracks divided in sectors
  - Inter-sector gaps
  - Sectors are typically formated to be 512 bytes-long
- Logical structure
  - Unidimensional, linear array of blocks
  - Block = 1 or *n* sectors
- Translation

```
blk = sec + #sec/tra x
(sur + cyl x #tra/cyl)
```



- Partition
  - Set of contiguous disk cylinders considered by the OS as an autonomous logical disk

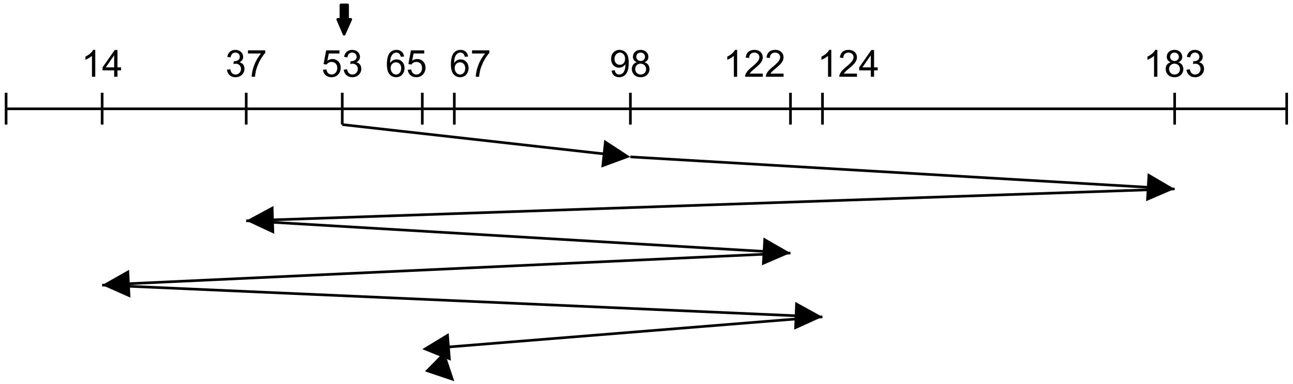
Disk Scheduling

- Disk access time parameters
  - Seek: time to move the arm to a given cylinder
  - Latency: delay until a sector passes under the head
  - Transfer: time to transfer data from the disk controller to main memory
- Disk access requests
  - Disk address + memory address + size
  - Request queue
    - Order requests gathering those for the same cylinder
    - Order requests for different cylinders to reduce seek time
- Other performance factors
  - File organization (contiguous/disperse)
  - Control info location
  - Cache

Disk Scheduling Algorithms

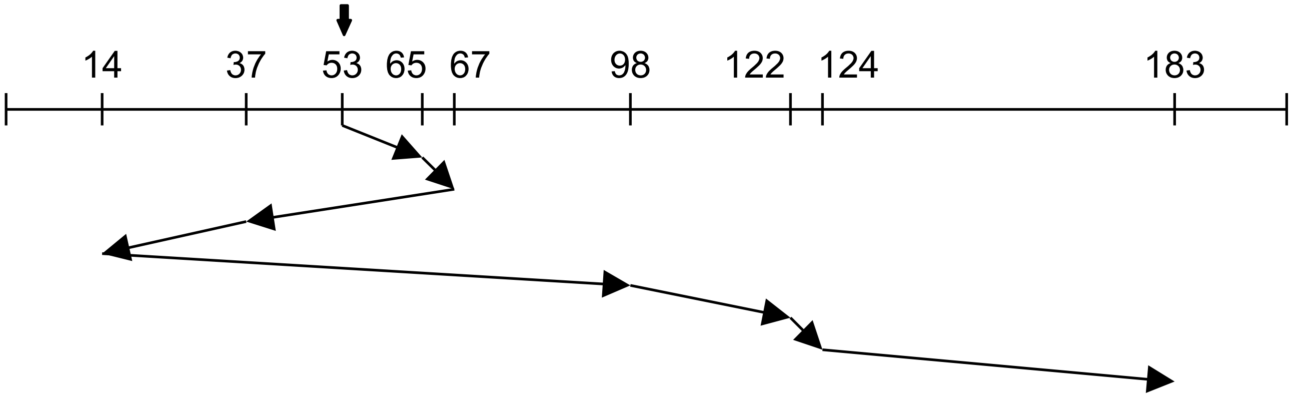
- First-Come First-Served (FCFS)

  Queue: 98, 183, 37, 122, 14, 124, 65, 67   *Seek*: 640 tracks



- Shortest Seek Time First (SSTF)

  Queue: 98, 183, 37, 122, 14, 124, 65, 67   *Seek*: 236 tracks

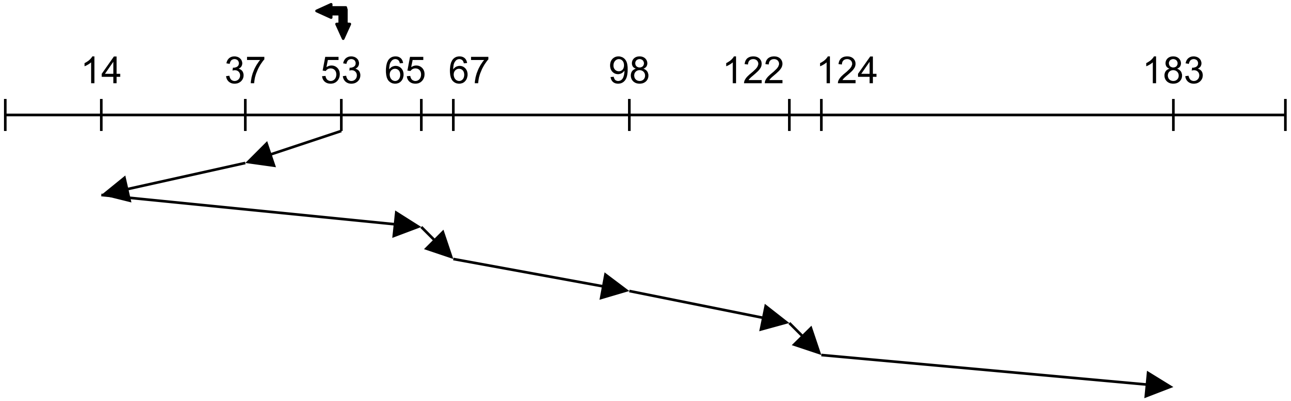


Operating Systems

Disk Scheduling Algorithms

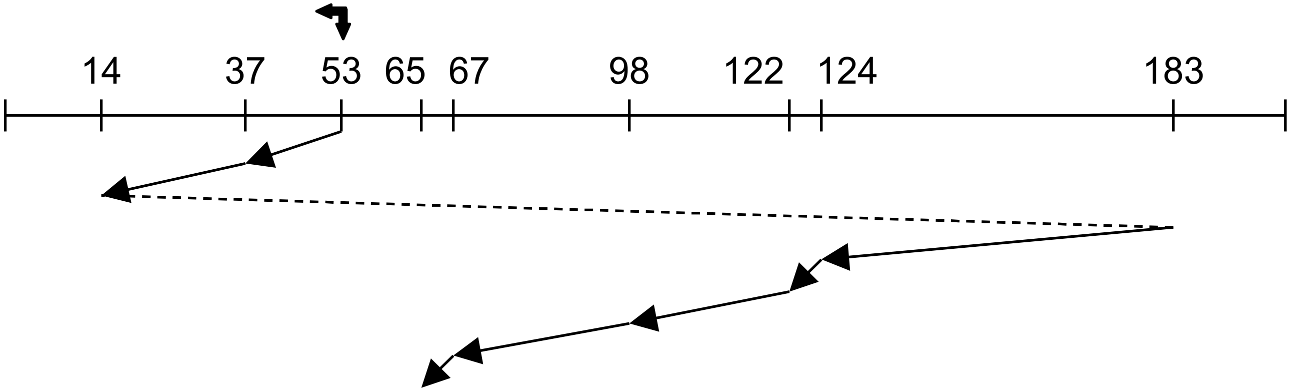
- Scan (Elevator)

Queue: 98, 183, 37, 122, 14, 124, 65, 67 *Seek*: 208 tracks



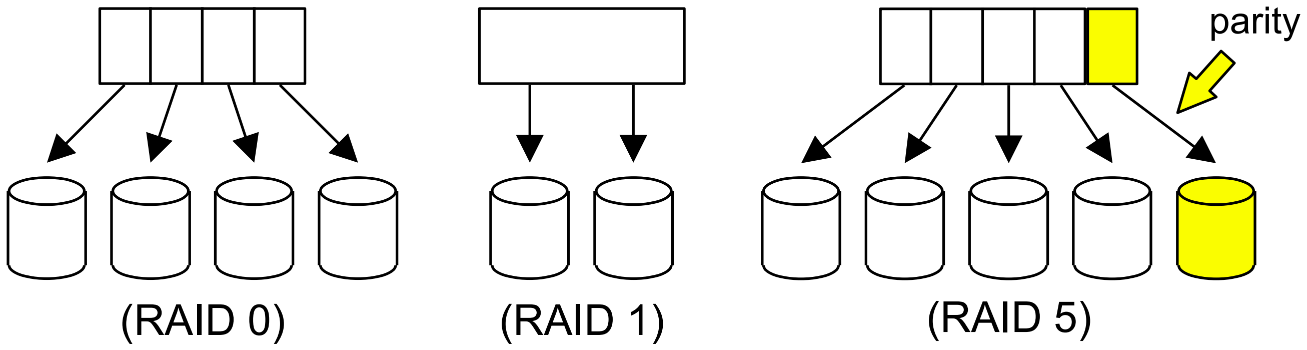
- Circular Scan (C-SCAN)

Queue: 98, 183, 37, 122, 14, 124, 65, 67 *Seek*: 326 tracks



Redundant Array of Independent Disks

- RAID 0 (stripping)
  - Each block is broken down in sub-blocks
  - Each sub-block is stored on a different disk
  - High performance
- RAID 1 (shadowing/mirroring)
  - Each block is stored twice
  - High reliability
- RAID 5 (stripping + rotating parity)
  - High performance with good reliability



I/O Management

- Interactive systems are often more concerned with I/O than computing
- I/O devices
  - Vary widely in functionality and speed
  - Standard software and hardware interfaces help to incorporate new devices
  - New devices are constantly introduced
- Device driver
  - Bridge between OS subsystems and I/O devices
  - Encapsulate device particularities delivering an uniform interface

I/O Hardware

- Port
  - Host connection point for I/O devices
- Bus
  - Shared set of wires and a protocol that allows several devices to be simultaneously connected to the host
- Controller
  - Controls the operation of ports, buses and devices
  - From simple electronics to complex processors
  - Interacts with host through registers
    - Control, status, data in/out
    - I/O ports, memory mapped, CPU register mapped

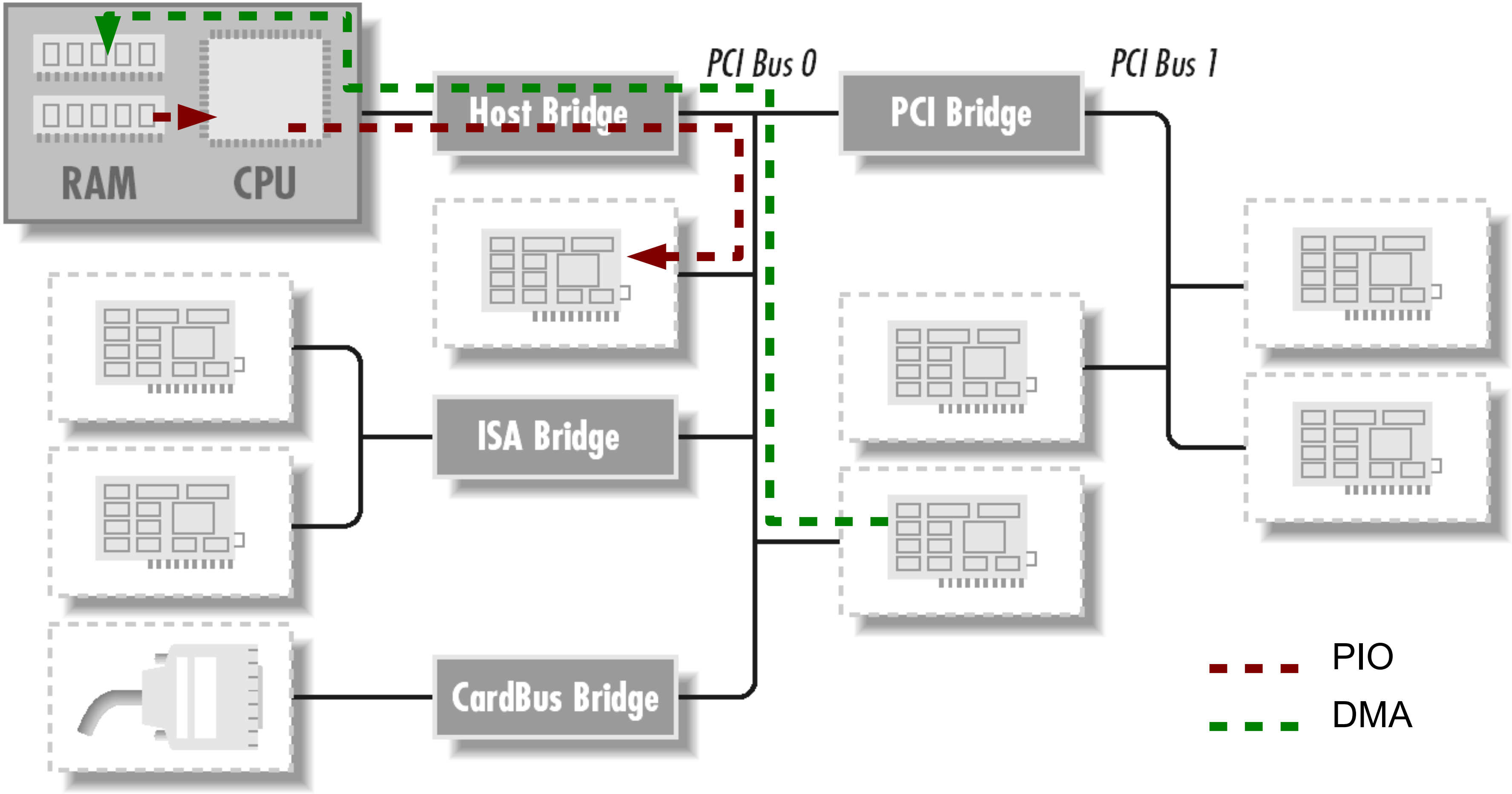
I/O Operation

- Polling
  - Host 'polls' status registers to determine the status of a device
  - Busy-waiting
    - Loop reading a status register
    - Overhead on multitask systems
    - Simplicity and efficiency on single-task systems
- Interrupts
  - Avoids busy-waiting
  - I/O device receives a service request and generates and interrupt when the request has been accomplished
  - Transparent to processes

I/O Data Transfers

- Programmed I/O
  - Data is transfered to/from I/O device by having the CPU to write/read data registers on the device controller
  - One word at a time
- Direct Memory Access (DMA)
  - Data is transfered by dedicated circuitry (DMA controller) without CPU assistance
    - Source and destination pointers + count
    - Multi-word (burst) transfers
    - Interrupt on completion or error
  - Concurs with CPU for memory
  - Pitfall
    - Address translation logical -> physical or DVMA

I/O Hardware



🟥 PIO
🟩 DMA

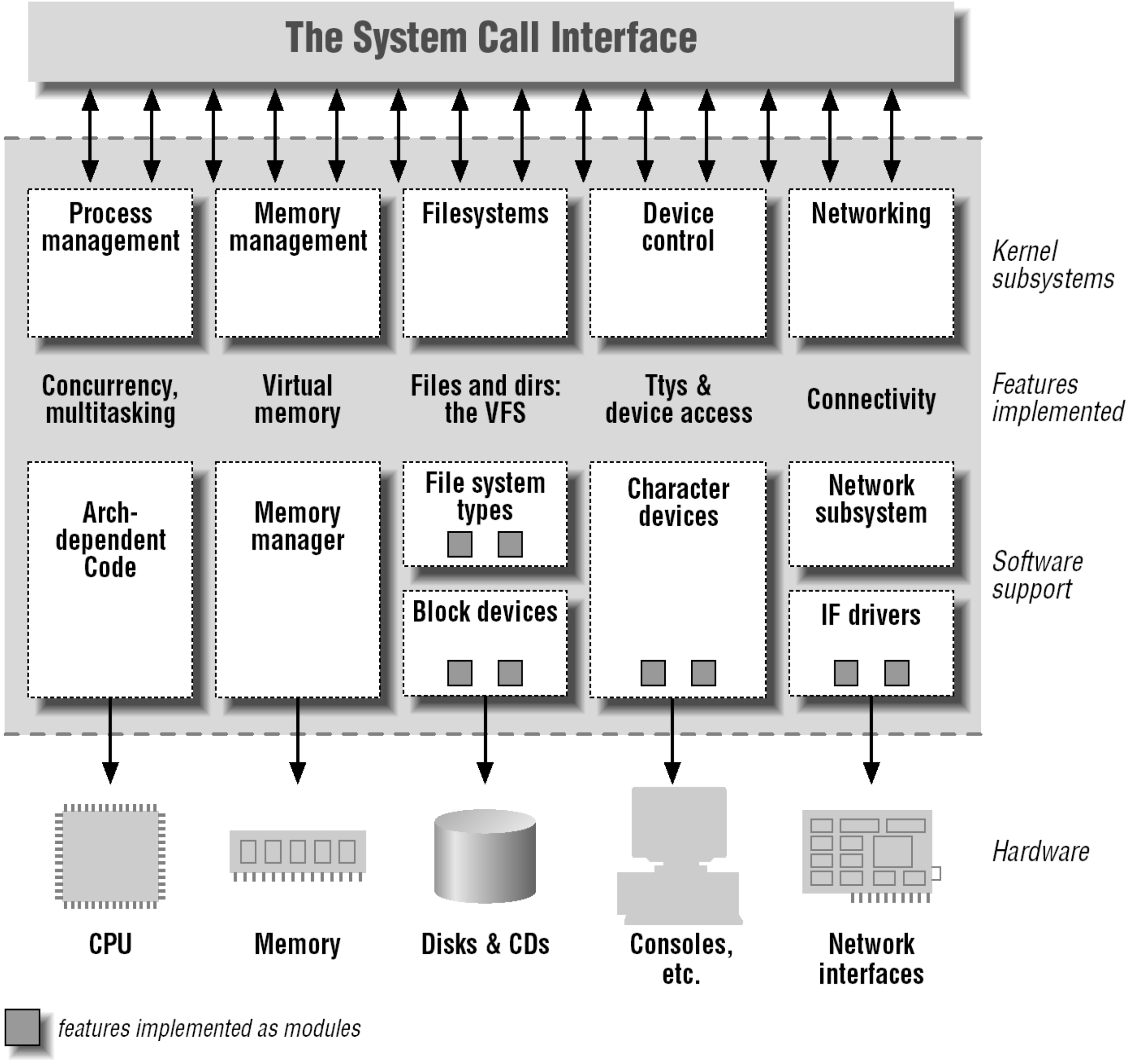
Application I/O Interface

- Indirect via I/O subsystems
  - A disk can be indirectly accessed through the files contained on it
  - A network adapter can be indirectly accessed through the TCP/IP stack (socket)
- Pseudo-file
  - Device drivers become handlers of operations on 'special files' that are plugged into the file system (/dev/mouse, /dev/hda, etc)
- Specific system calls
  - OS provides specific system calls to interact with I/O devices (eg ioctl on Unix)

Unix (Linux) Device Drivers

- Kernel module that handles the interaction with an specific hardware device, hiding its operational details behind a common interface
- Three basic categories
  - Character
  - Block
  - Network

Kernel Overview: LINUX



Hardware Devices

- Accessible via `/dev` pseudo-files
- Kernel redirect pseudo-file operations to proper device driver services considering *major* and *minor* numbers
  - *Major*
    - Identifies a driver within the kernel (8 bits)
  - *Minor*
    - Identifies a device (unit) within the driver

Char Devices

- Byte streams (e.g. `/dev/console`, `/dev/ttyS0`, `/dev/st0`)
- Operate mostly like ordinary files
  - No backward seeks

Block Devices

- Block-accessible devices at I/O level
- File system related devices (e.g. disks)
- Share a common interface with char devices, but distinct semantics
  - Block oriented (accessing single bytes is a waste)
  - Seekable
- Additional operations to support file systems

Net Devices

- Do not fit properly under the pseudo-file interface
  - Usually not a node in a file system
  - Integration with a protocol stack
- Generic network interface instead
  - Communication related operations (e.g. sending, receiving, package marshaling, time-out handling, statistic collection)
  - Optimized for TCP/IP integration

Hello World Module

```
[root]#cat > hello.c
#define MODULE
#include <linux/module.h>
int init_module(){printk(“Hello World!”); return 0;}
void cleanup_module(){printk(“Good Bye!”);}
^D
[root]# gcc -c hello.c
[root]# insmod hello.o
[root]# dmesg
[root]# rmmod hello
[root]# dmesg
```

Module Initialization

- Initialization

```
int init_module(void)
```

  - Module's entry point
  - Called at loading (by insmod)
  - Performs module registration
- Finalization

```
void cleanup_module(void)
```

  - Module's exit point
  - Called at unloading (by rmmod)
  - Performs module unregistration

Module Registration

- Binds a module to the kernel's syscall interface
- Registration

```
int register_chrdev(unsigned int major, const
char *name, struct file_operations *fops)
```

- Unregistration

```
int unregister_chrdev(unsigned int major,
const char *name)
```

- Pseudo file

```
mknod /dev/devname0 c major minor
```

Module Parameters

- Externally accessible module-global variables
- Declared via MODULE_PARM macro

```
int irq = 10;
char * name = “Unknown”;
MODULE_PARM(irq,”i”); /* declare irq as int */
MODULE_PARM(name,”s”); /* declare name as string
  */
```

- Defined at load time

```
insmod mod.o irq=9 name=.”The Server”
```

Module Info

- Externally visible module declarations used to supply clients with some useful information
- Macros

```
MODULE_AUTHOR("Somebody");
MODULE_DESCRIPTION("This module doesn't do
  anything");
MODULE_PARM_DESC(irq, "Device IRQ (3/4)"
```

struct file_operations

```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char *, size_t, ...
  ssize_t (*write) (struct file *, const char *, ...
  int (*readdir) (struct file *, void *, filldir_t);
  unsigned int (*poll) (struct file *, struct ...
  int (*ioctl) (struct inode *, struct file *, ...
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, struct dentry *, ...
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
  :
  :
};
```

struct file

```
struct file {
  struct list_head        f_list;
  struct dentry           *f_dentry;
  struct vfsmount         *f_vfsmnt;
  struct file_operations  *f_op;
  atomic_t                f_count;
  unsigned int            f_flags;
  mode_t                  f_mode;
  loff_t                  f_pos;
  unsigned long           f_reada, f_ramax, f_raend,
                                f_ralen, f_rawin;
  struct fown_struct      f_owner;
  unsigned int            f_uid, f_gid;
  int                     f_error;
  :
};
```

```
MINOR(f_dentry->d_inode->i_rdev)
```

Module's Reference Counter

- Automatically tracks how many clients a module has at a moment
- Avoids unloading a module that is being used by a client
- Manipulated by macros

```
MOD_INC_USE_COUNT
MOD_DEC_USE_COUNT
MOD_IN_USE
```

Programming Hits

- No standard libraries (and headers)
  - `printk` instead of `printf`

```
#include <linux/x.h>
#include <asm/y.h>
```

- Signalize kernel code

```
#define __KERNEL__
```

- Avoid name-clashes
  - Local symbols (`static`)
  - Prefixed symbols (`mod_sym`)

```
EXPORT_NO_SYMBOLS;
```

More Programming Hits

- Kernel code runs within the context of calling user process

```
#include <asm/uaccess.h>
unsigned long copy_to_user(to, from, count);
unsigned long copy_from_user(to, from, count);
```

- In-kernel memory allocation

```
#include <linux/malloc.h>
void *kmalloc(unsigned int size, int priority);
void kfree(void *obj);
```

Distributed Systems Taxonomy

- Stand-alone computing systems
  - Independent computers
  - Independent tasks
- Networked computing systems
  - Interconnected independent computers
  - Processes of independent tasks can communicate
- Distributed computing systems
  - Loosely-coupled computers
  - Processes of individual tasks transparently share resources
- Parallel computing systems
  - Tightly-coupled processing units
  - Several processes cooperate on a single task

A New Perspective

- Computing systems are merging
  - Embedded systems were once stand-alone
    - Now modern limousines are distributed systems on wheels
  - Workstations were once networked systems
    - They now use parallel hardware (e.g. SMP, GPU)
    - Transparency is increasing (e.g. peer-to-peer)
  - Distributed systems were once local
    - Now the web is the computer (SETI@Home)
  - Parallel systems were once run on supercomputers
  - Clusters are now made of off-the-shelf computers with high-speed buses and networks
- Operating systems are being challenged
  - Light enough to support a stand-alone system
  - Powerful enough to support a distributed system
  - And parallelism on both cases

Distributed Systems

- Set of loosely coupled computers interconnected by a network
- Each computer has its own local resources plus remote resources from other computers in the set
- Processes on a distributed system access resources independently of whether they are local or remote (location transparency)
- Inter-process communication is mostly based on message passing
- Process models
  - Client-Server
    - Server has a resource that is used by the client
  - Peer-to-Peer
    - Both partner processes share some of their resources

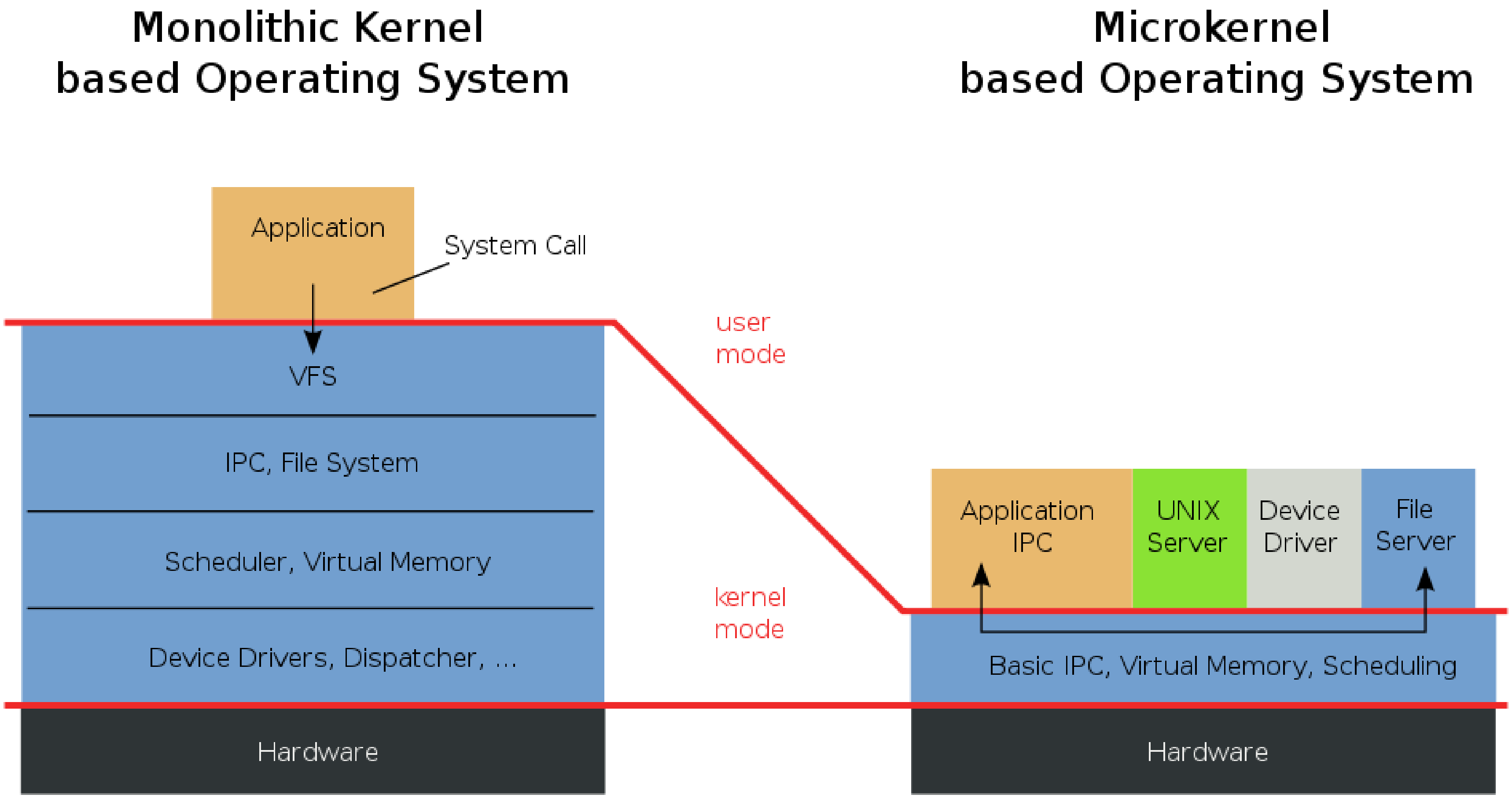
Motivation

- Resource sharing
  - Remote file sharing, printing, access to special devices (scanner, CD writer, etc)
  - Distributed databases
- Computation speedup
  - Tasks can be partitioned and distributed
- Reliability
  - The failure of a node does not necessarily disrupts the system
- Scalability
  - New nodes can be aggregated to the system on demand
- Pitfalls: complexity and security

Transparency

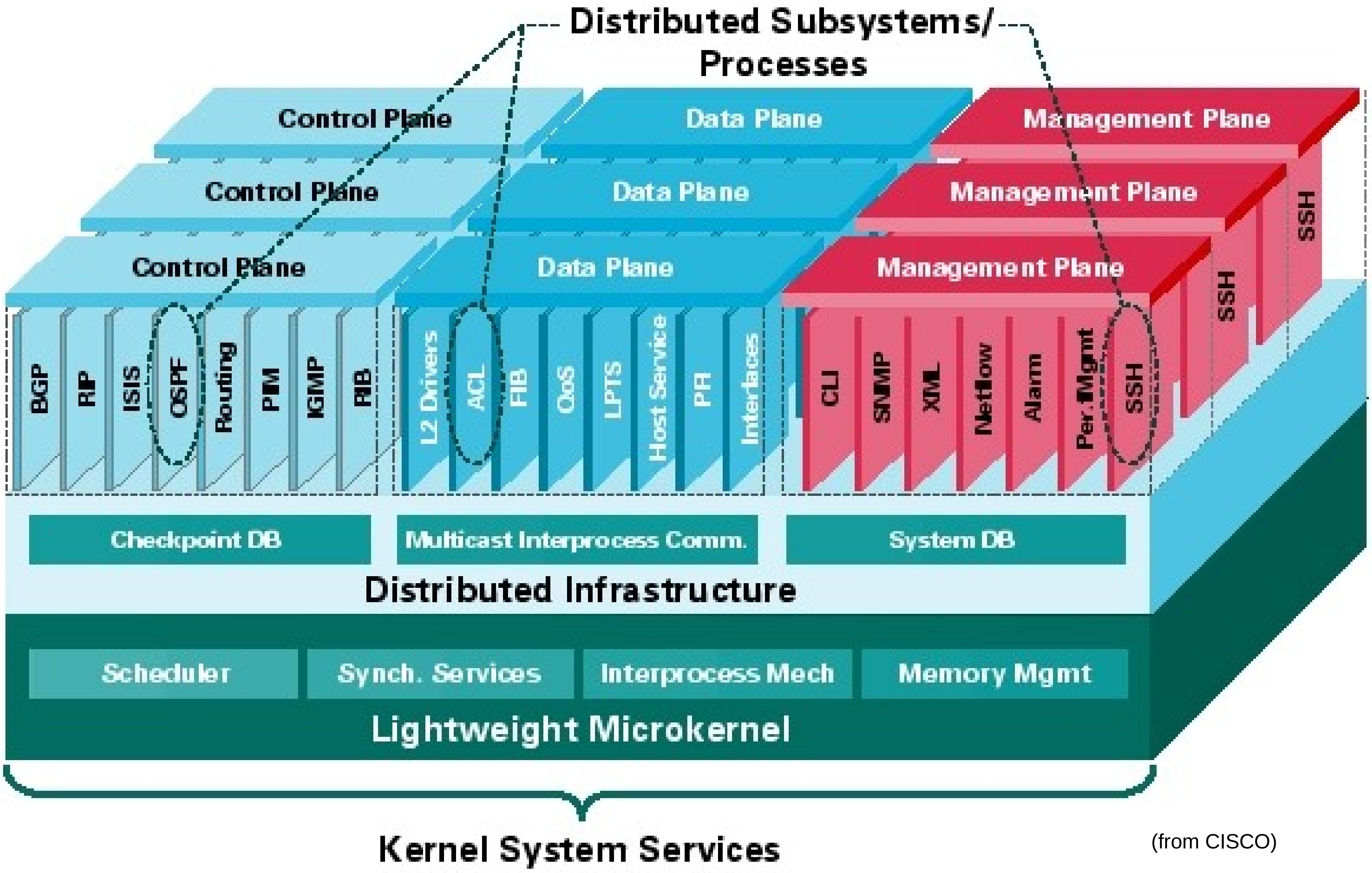
- Location transparency
  - Local and remote objects look just the same
  - No need to specify location
- Migration transparency
  - Objects change location, their names are preserved
- Replication transparency
  - Objects can be automatically replicated (consistency)
- Concurrency transparency
  - Objects can be concurrently manipulated without explicit synchronization
- Parallelism transparency
  - Automatic parallelization

μ-kernels and Distributed Systems



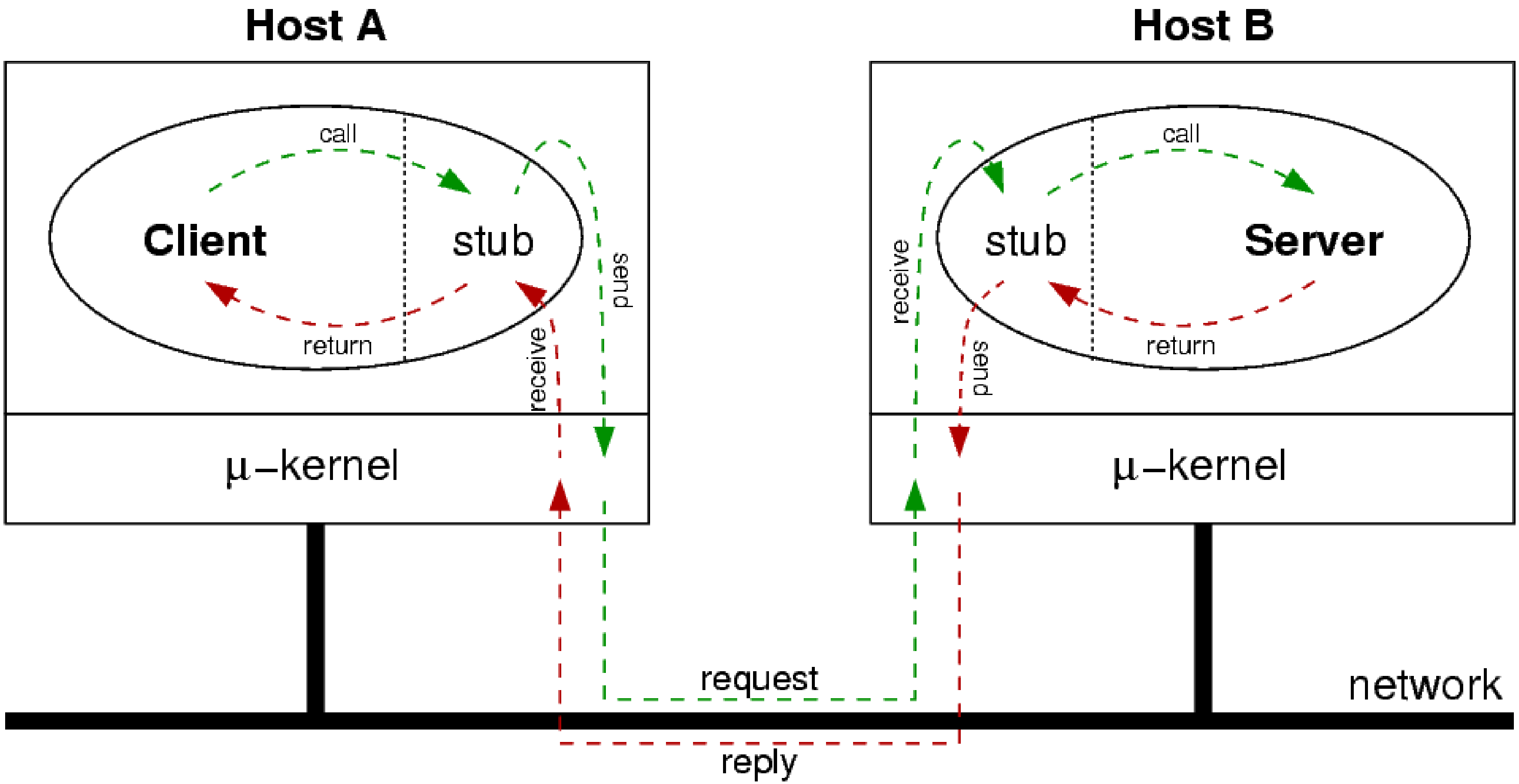
(from Wikipedia)

CISCO IOS XR



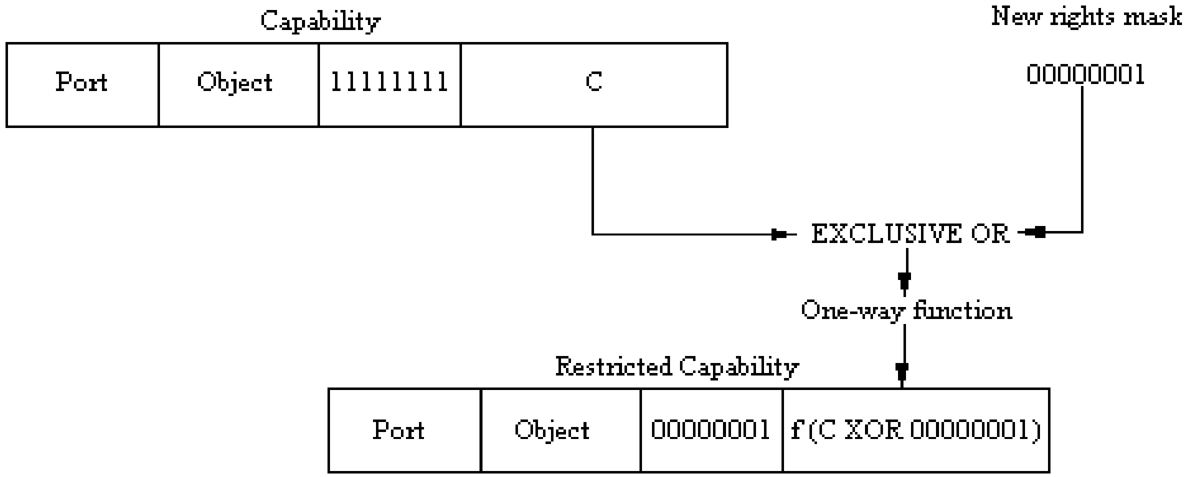
(from CISCO)

Remote Procedure Call (RPC)

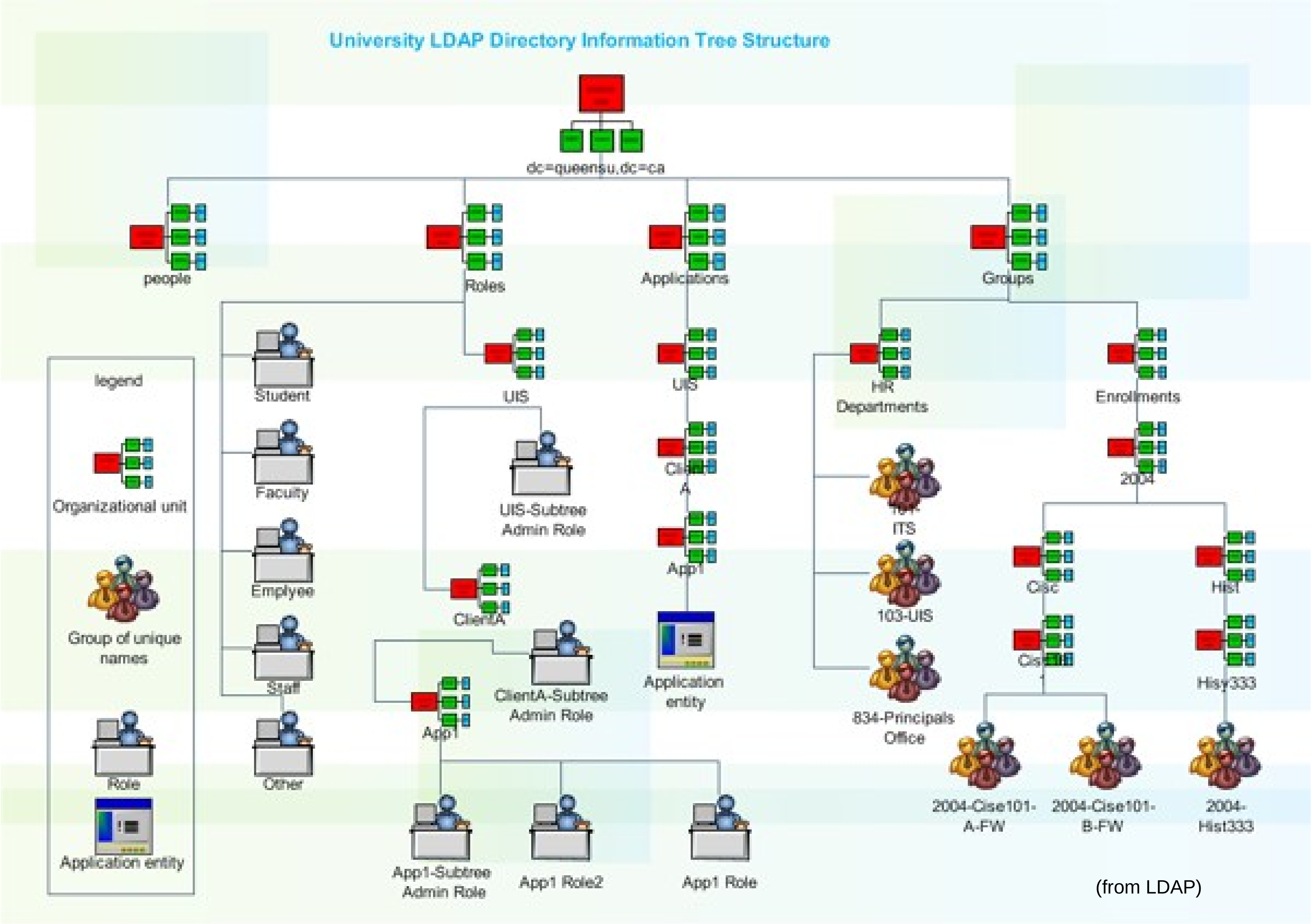


Object Identification

- Implicit id (domain specific)
  - Object pointer
- Local id
  - Object counter with reuse and overflow control
- Global id
  - Host id + local id
  - Easy on a MAC-assigned, IP-based world
- Capability
  - Global id + permissions + secret

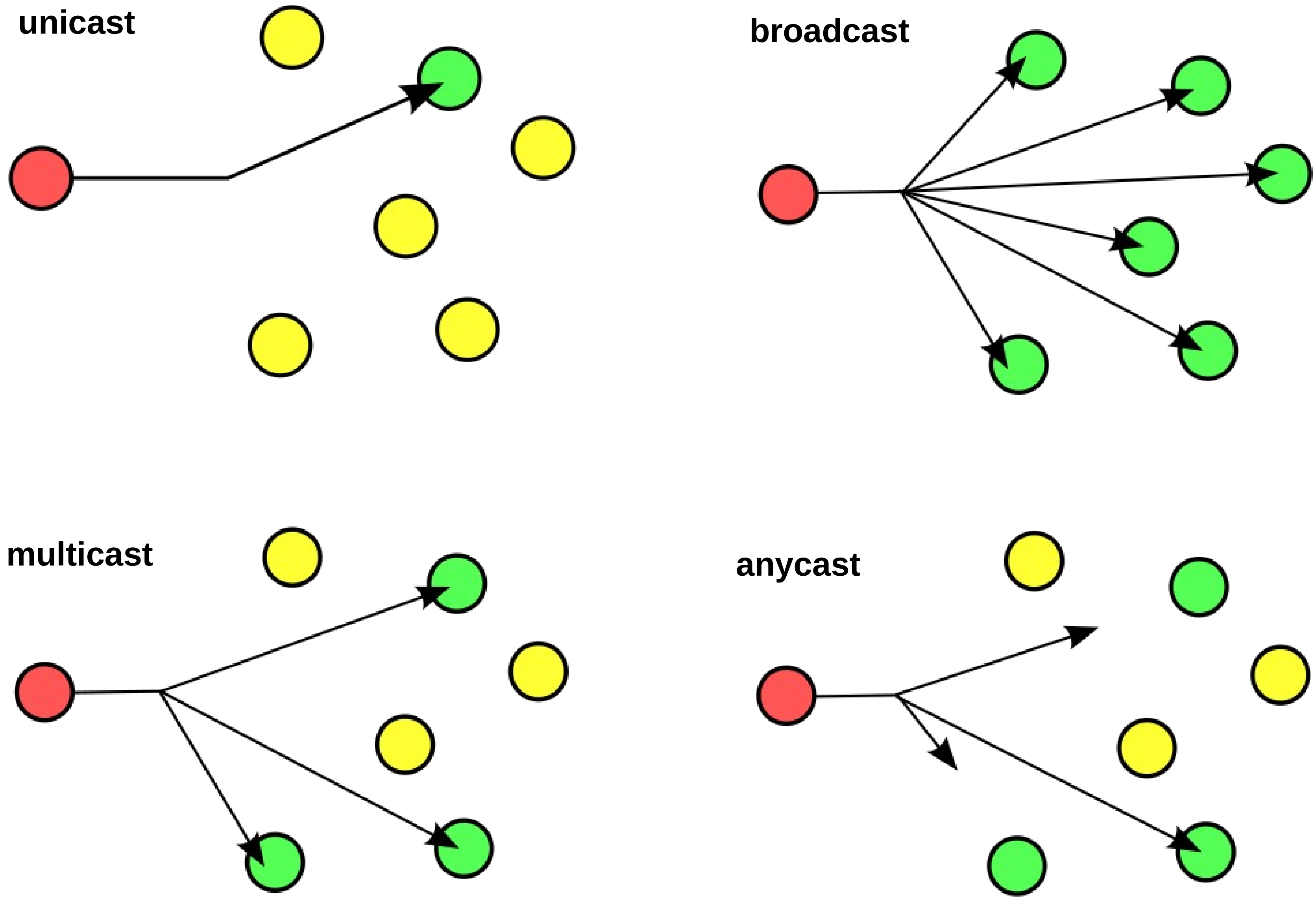


Name Server



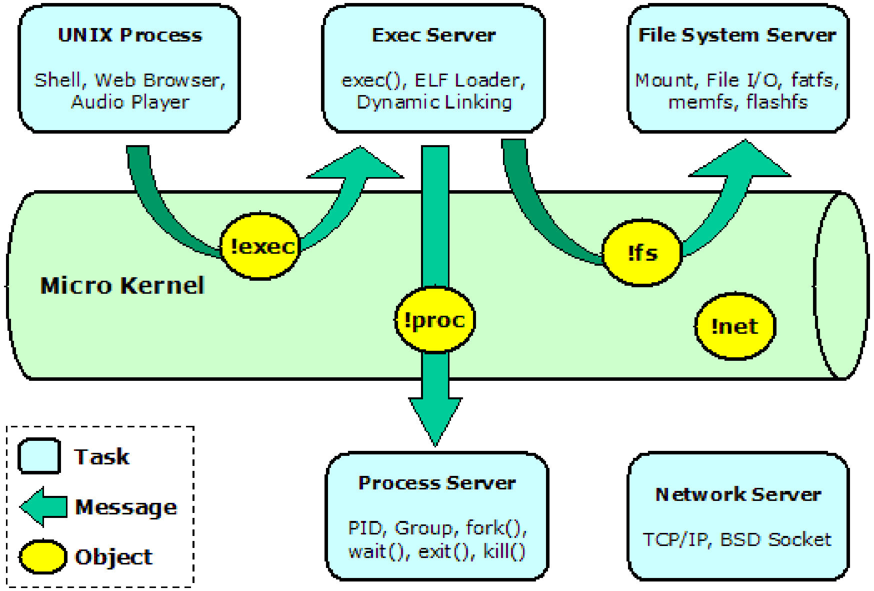
(from LDAP)

Communication Patterns



(from Wikipedia)

Inter-process Communication


UNIX Process
Shell, Web Browser, Audio Player
Exec Server
exec(), ELF Loader, Dynamic Linking
File System Server
Mount, File I/O, fatfs, memfs, flashfs
!exec
!fs
Micro Kernel
!proc
!net
Task
Message
Object
Process Server
PID, Group, fork(), wait(), exit(), kill()
Network Server
TCP/IP, BSD Socket

- Messages used to request u-kernel services can be forwarded to other hosts
  - Global id
  - Network driver/service
- Foundation for all other distributed system services