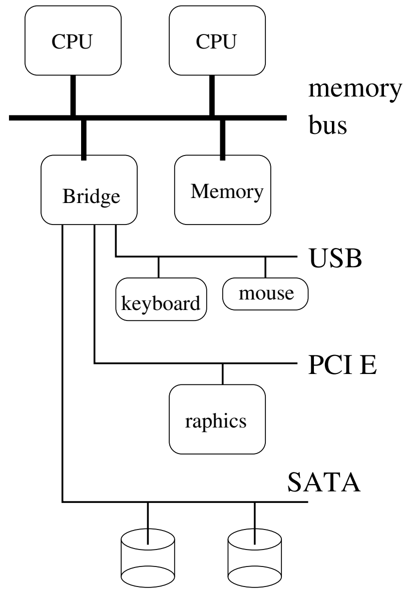
CS 350
Operating Systems
Course Notes (Part 2)

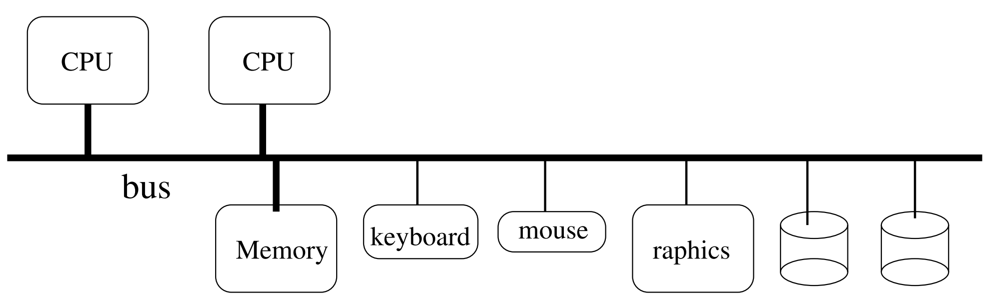
Spring 2014

David R. Cheriton
School of Computer Science
University of Waterloo

Bus Architecture Example



Simplified Bus Architecture



Sys/161 LAMEbus Device Examples

- timer/clock - current time, timer, beep
- disk drive - persistent storage
- serial console - character input/output
- text screen - character-oriented graphics
- network interface - packet input/output

Device Register Example: Sys/161 timer/clock

Offset	Size	Type	Description
0	4	status	current time (seconds)
4	4	status	current time (nanoseconds)
8	4	command	restart-on-expiry
12	4	status and command	interrupt (reading clears)
16	4	status and command	countdown time (microseconds)
20	4	command	speaker (causes beeps)

Device Register Example: Sys/161 disk controller

Offset	Size	Type	Description
0	4	status	number of sectors
4	4	status and command	status
8	4	command	sector number
12	4	status	rotational speed (RPM)
32768	512	data	transfer buffer

Device Drivers

- a device driver is a part of the kernel that interacts with a device
- example: write character to serial output device

```
write character to device data register
write output command to device command register
repeat {
    read device status register
} until device status is ''completed''
clear the device status register
```

- this example illustrates *polling*: the driver repeatedly checks whether the device is finished, until it is finished.

Another Polling Example

```
write target sector number into sector number register
write output data (512 bytes) into transfer buffer
write ``write'' command into status register
repeat {
   read status register
} until status is ``completed'' (or error)
clear the status register
```

Disk operations are slow. The device driver may have to poll for a long time.

Using Interrupts to Avoid Polling

- polling can be avoided if the device can use interrupts to indicate that it is finished
- example: disk write operation using interrupts:

```
write target sector number into sector number register
write output data (512 bytes) into transfer buffer
write ''write'' command into status register
block until device generates completion interrupt
read status register to check for errors
clear status register
```

- while thread running the driver is blocked, the CPU is free to run other threads
- kernel synchronization primitives (e.g., semaphores) can be used to implement blocking

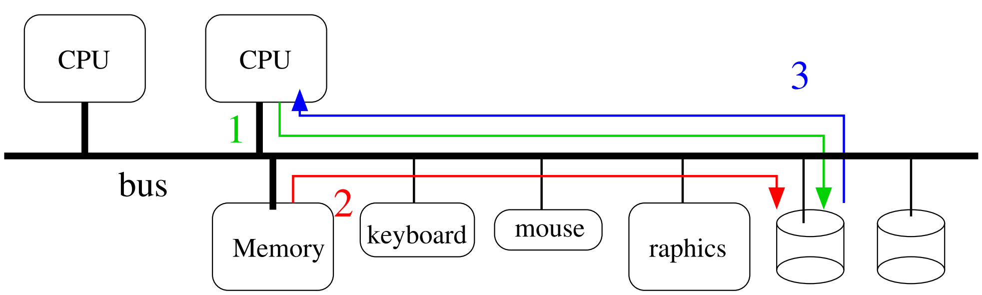
Device Data Transfer

- Sometimes, a device operation will involve a large chunk of data - much larger than can be moved with a single instruction.
  - example: disk read or write operation
- Devices may have data buffers for such data - but how to get the data between the device and memory?
  - Option 1: *program-controlled I/O*
    The device driver moves the data iteratively, one word at a time.
    - Simple, but the CPU is *busy* while the data is being transferred.
  - Option 2: *direct memory access (DMA)*
    - CPU is *not busy* during data transfer, and is free to do something else.

Sys/161 LAMEbus devices do program-controlled I/O.

Direct Memory Access (DMA)

- DMA is used for block data transfers between devices (e.g., a disk controller) and memory
- Under DMA, the CPU initiates the data transfer and is notified when the transfer is finished. However, the device (not the CPU) controls the transfer itself.



1. CPU issues DMA request to controller
2. controller directs data transfer
3. controller interrupts CPU

Device Driver for Disk Write with DMA

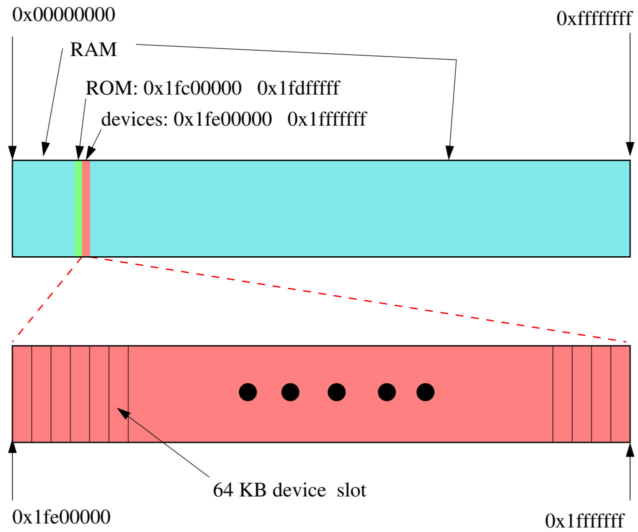
```
write target disk sector number into sector number register
write source memory address into address register
write ''write'' command into status register
block (sleep) until device generates completion interrupt
read status register to check for errors
clear status register
```

Note: driver no longer copies data into device transfer buffer

Accessing Devices

- how can a device driver access device registers?
- Option 1: special I/O instructions
  - such as `in` and `out` instructions on x86
  - device registers are assigned "port" numbers
  - instructions transfer data between a specified port and a CPU register
  -
- Option 2: memory-mapped I/O
  - each device register has a physical memory address
  - device drivers can read from or write to device registers using normal load and store instructions, as though accessing memory

MIPS/OS161 Physical Address Space

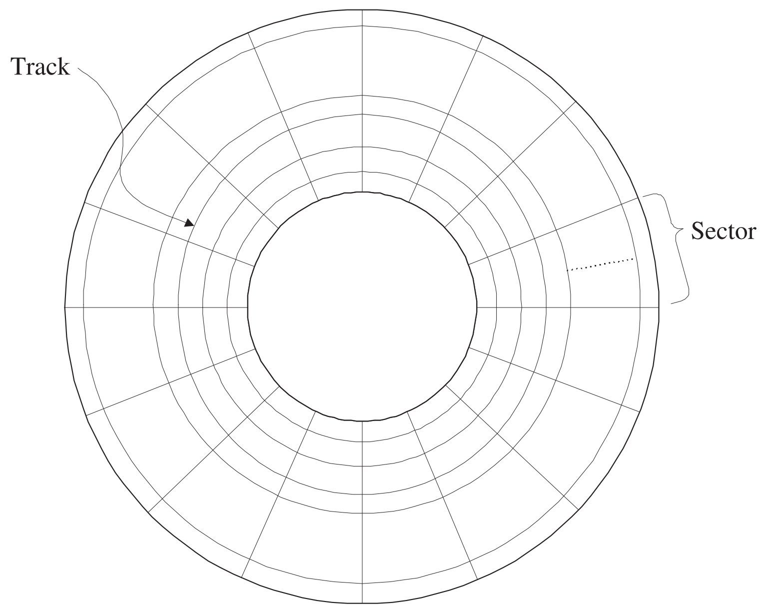


Each device is assigned to one of 32 64KB device “slots”. A device’s registers and data buffers are memory-mapped into its assigned slot.

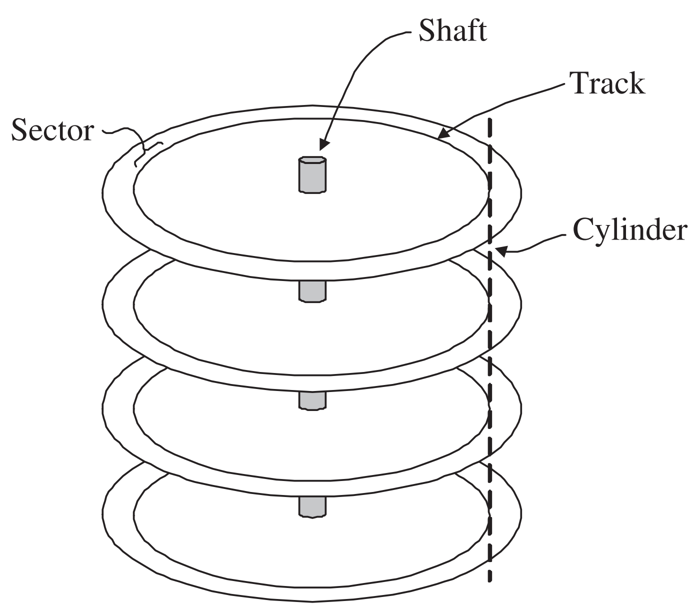
Logical View of a Disk Drive

- disk is an array of numbered blocks (or sectors)
- each block is the same size (e.g., 512 bytes)
- blocks are the unit of transfer between the disk and memory
  - typically, one or more contiguous blocks can be transferred in a single operation
- storage is *non-volatile*, i.e., data persists even when the device is without power

A Disk Platter's Surface



Physical Structure of a Disk Drive



Simplified Cost Model for Disk Block Transfer

- moving data to/from a disk involves:

  seek time: move the read/write heads to the appropriate cylinder

  rotational latency: wait until the desired sectors spin to the read/write heads

  transfer time: wait while the desired sectors spin past the read/write heads
- request service time is the sum of seek time, rotational latency, and transfer time

$$t_{service} = t_{seek} + t_{rot} + t_{transfer}$$

- note that there are other overheads but they are typically small relative to these three

Rotational Latency and Transfer Time

- rotational latency depends on the rotational speed of the disk
- if the disk spins at ω rotations per second:

$$0 \leq t_{rot} \leq \frac{1}{\omega}$$

- expected rotational latency:

$$\bar{t}_{rot} = \frac{1}{2\omega}$$

- transfer time depends on the rotational speed and on the amount of data transferred
- if k sectors are to be transferred and there are T sectors per track:

$$t_{transfer} = \frac{k}{T\omega}$$

Seek Time

- seek time depends on the speed of the arm on which the read/write heads are mounted.
- a simple linear seek time model:
  - $t_{maxseek}$ is the time required to move the read/write heads from the innermost cylinder to the outermost cylinder
  - C is the total number of cylinders
- if k is the required *seek distance* ($k > 0$):

$$t_{seek}(k) = \frac{k}{C} t_{maxseek}$$

Performance Implications of Disk Characteristics

- larger transfers to/from a disk device are *more efficient* than smaller ones. That is, the cost (time) per byte is smaller for larger transfers. (Why?)
- sequential I/O is faster than non-sequential I/O
  - sequential I/O operations eliminate the need for (most) seeks
  - disks use other techniques, like *track buffering*, to reduce the cost of sequential I/O even more

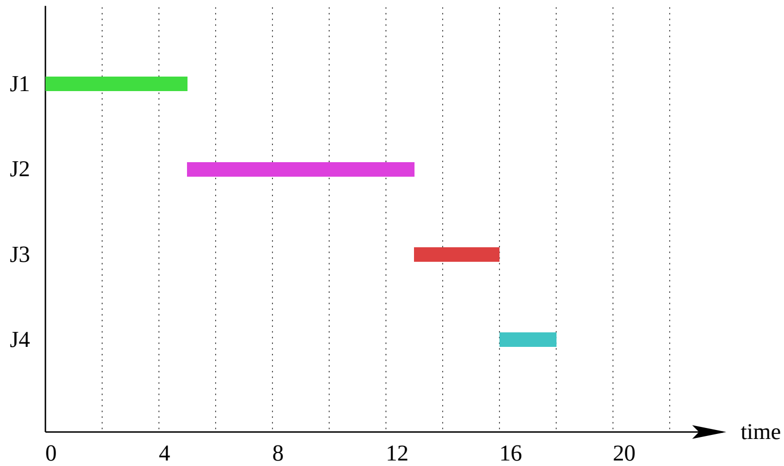
Job Scheduling Model

- problem scenario: a set of *jobs* needs to be executed using a single server, on which only one job at a time may run
- for the ith job, we have an arrival time a_i and a run time r_i
- after the ith job has run on the server for total time r_i, it finishes and leaves the system
- a job *scheduler* decides which job should be running on the server at each point in time
- let s_i ($s_i \geq a_i$) represent the time at which the ith job first runs, and let f_i represent the time at which the ith job finishes
  - the *turnaround time* of the ith job is $f_i - a_i$
  - the *response time* of the ith job is $s_i - a_i$

Basic Non-Preemptive Schedulers: FCFS and SJF

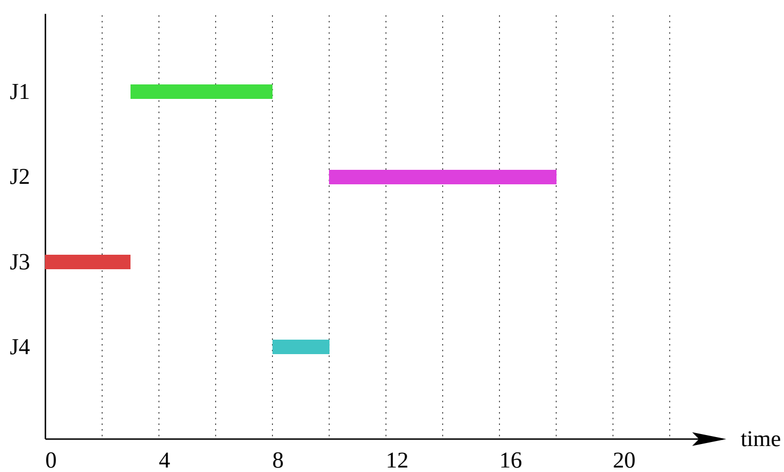
- FCFS: runs jobs in arrival time order.
  - simple, avoids starvation
  - pre-emptive variant: round-robin
- SJF: shortest job first - run jobs in increasing order of r_i
  - minimizes average *turnaround* time
  - long jobs may starve
  - pre-emptive variant: SRTF (shortest remaining time first)

FCFS Gantt Chart Example



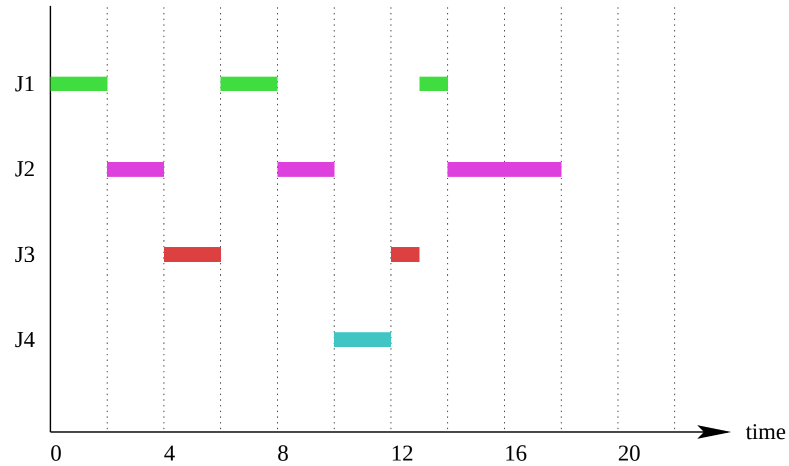
Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	2
run time (r_i)	5	8	3	2

SJF Example



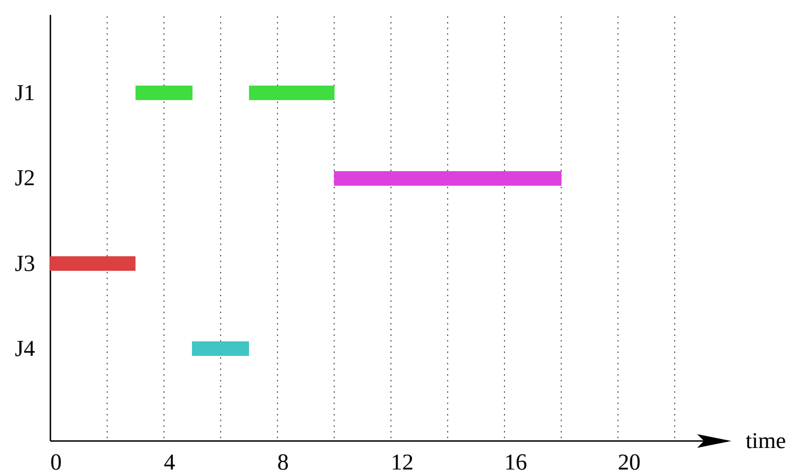
Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	2
run time (r_i)	5	8	3	2

Round Robin Example



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	2
run time (r_i)	5	8	3	2

SRTF Example



Job	J1	J2	J3	J4
arrival (a_i)	0	0	0	2
run time (r_i)	5	8	3	2

CPU Scheduling

- CPU scheduling is job scheduling where:
  - the server is a CPU (or a single core of a multi-core CPU)
  - the jobs are *ready threads*
    - a thread "arrives" when it becomes ready, i.e., when it is first created, or when it wakes up from sleep
    - the run-time of the thread is the amount of time that it will run before it either finishes or blocks
  - thread run times are typically *not known* in advance by the scheduler
- typical scheduler objectives
  - responsiveness - low *response time* for some or all threads
  - "fair" sharing of the CPU
  - efficiency - there is a cost to switching

Prioritization

- CPU schedulers are often expected to consider process or thread priorities
- priorities may be
  - specified by the application (e.g., Linux `setpriority/sched_setscheduler`)
  - chosen by the scheduler
  - some combination of these
- two approaches to scheduling with priorites
  1. schedule the highest priority thread
  2. weighted fair sharing
     - let p_i be the priority of the ith thread
     - try to give each thread a "share" of the CPU in proportion to its priority:

$$\frac{p_i}{\sum_j p_j} \tag{1}$$

Multi-level Feedback Queues

- objective: good responsiveness for *interactive* processes
  - threads of interactive processes block frequently, have short run times
- idea: gradually diminish priority of threads with long run times and infrequent blocking
  - if a thread blocks before its quantum is used up, *raise* its priority
  - if a thread uses its entire quantum, *lower* its priority

Multi-level Feedback Queues (Algorithm)

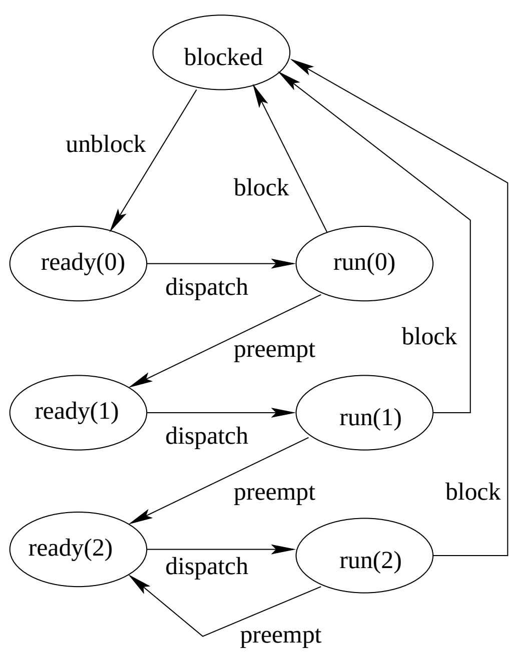
- scheduler maintains several round-robin ready queues
  - highest priority threads in queue Q_0, lower priority in Q_1, still lower in Q_2, and so on.
- scheduler always chooses thread from the lowest non-empty queue
- threads in queue Q_i use quantum q_i, and $q_i \leq q_j$ if $i < j$
- newly ready threads go into ready queue Q_0
- a level i thread that is preempted goes into queue Q_{i+1}

This basic algorithm may *starve* threads in lower queues. Various enhancements can avoid this, e.g, periodically migrate all threads into Q_0.

Multilevel Feedback Queues

- objective: good responsiveness for *interactive* processes
  - threads of interactive processes block frequently, have short run times
- idea: gradually diminish priority of threads with long run times and infrequent blocking
- algorithm:
  - scheduler maintains several ready queues
  - scheduler never chooses a thread in ready queue i if there are threads in any ready queue $j < i$.
  - threads in ready queue i use quantum q_i, and $q_i \leq q_j$ if $i < j$
  - newly ready threads go into ready queue q_0
  - a level i thread that is preempted goes into the level $i + 1$ ready queue
  - plus: some rule for raising the priority of threads in lower queues

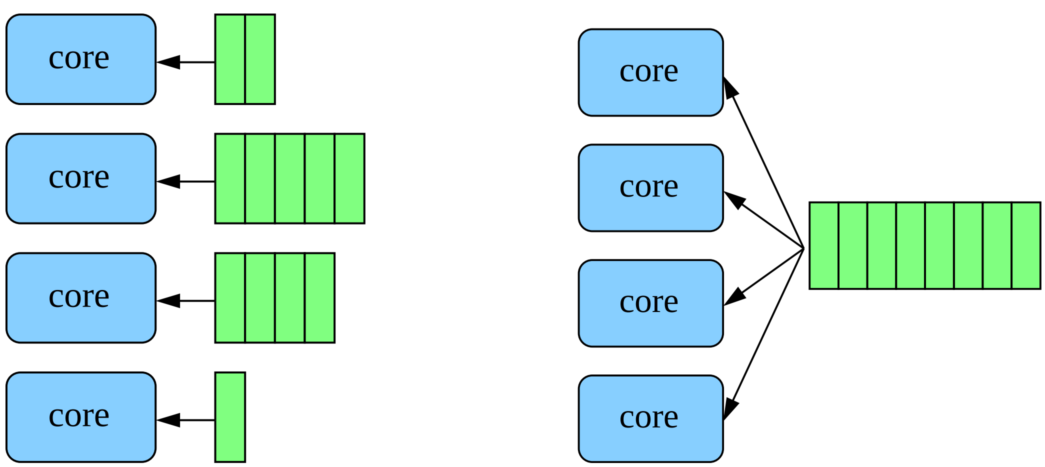
3 Level Feedback Queue State Diagram



Linux CFQ Scheduler - Key Ideas

- "Completely Fair Queueing" - a weighted fair sharing approach
- suppose that c_i is the actual amount of time that the scheduler has allowed the ith thread to run.
- on an *ideally shared* processor, we would expect $c_0 \frac{\sum_j p_j}{p_0} = c_1 \frac{\sum_j p_j}{p_1} = \cdots$
- CFQ calls $c_i \frac{\sum_j p_j}{p_i}$ the *virtual runtime* of the ith thread, and tracks it for each thread
- CFQ chooses the thread with the lowest virtual runtime, and runs it until some other thread's virtual runtime is lower (subject to a minimum runtime quantum)
  - virtual runtime advances more slowly for higher priority threads, so they get longer time slices
  - all ready threads run regularly, so good responsiveness

Scheduling on Multi-Core Processors



per core ready queue(s) vs. shared ready queue(s)

Scalability and Cache Affinity

- Contention and Scalability
  - access to shared ready queue is a critical section, mutual exclusion needed
  - as number of cores grows, contention for ready queue becomes a problem
  - per core design *scales* to a larger number of cores
- CPU cache affinity
  - as thread runs, data it accesses is loaded into CPU cache(s)
  - moving the thread to another core means data must be reloaded into that core's caches
  - as thread runs, it acquires an *affinity* for one core because of the cached data
  - per core design benefits from affinity by keeping threads on the same core
  - shared queue design does not

Load Balancing

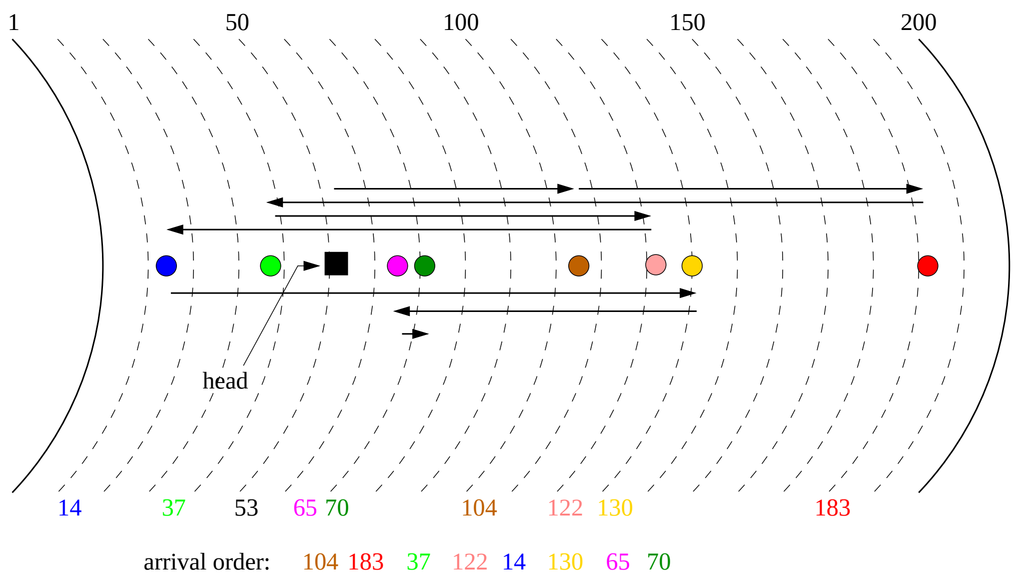
- in per-core design, queues may have different lengths
- this results in *load imbalance* across the cores
  - cores may be idle while others are busy
  - threads on lightly loaded cores get more CPU time than threads on heavily loaded cores
- not an issue in shared queue design
- per-core designs typically need some mechanism for *thread migration* to address load imbalances
  - migration means moving threads from heavily loaded cores to lightly loaded cores

Disk Head Scheduling

- goal: reduce seek times by controlling the order in which requests are serviced
- disk head scheduling may be performed by the controller, by the operating system, or both
- for disk head scheduling to be effective, there must be a queue of outstanding disk requests (otherwise there is nothing to reorder)
- an on-line approach is required: the disk request queue is not static

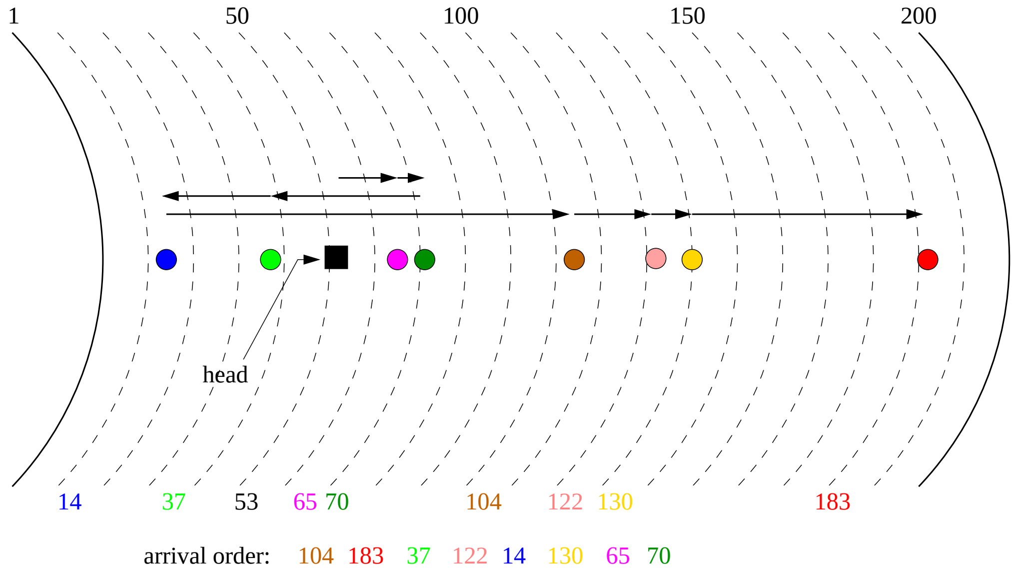
FCFS Disk Head Scheduling

- handle requests in the order in which they arrive
- fair and simple, but no optimization of seek times



Shortest Seek Time First (SSTF)

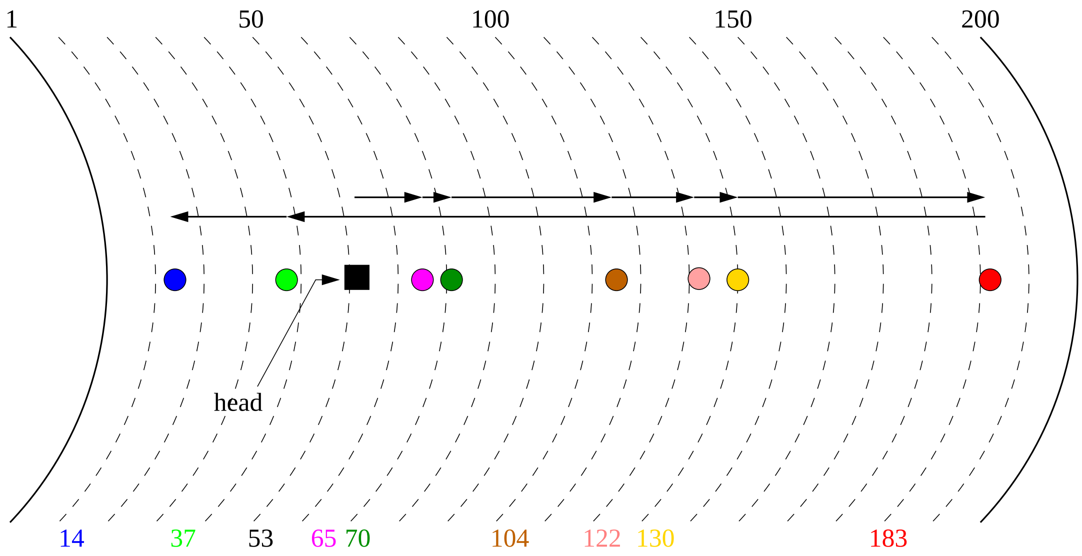
- choose closest request (a greedy approach)
- seek times are reduced, but requests may starve



Elevator Algorithms (SCAN)

- Under SCAN, aka the elevator algorithm, the disk head moves in one direction until there are no more requests in front of it, then reverses direction.
- there are many variations on this idea
- SCAN reduces seek times (relative to FCFS), while avoiding starvation

SCAN Example



arrival order: 104 183 37 122 14 130 65 70

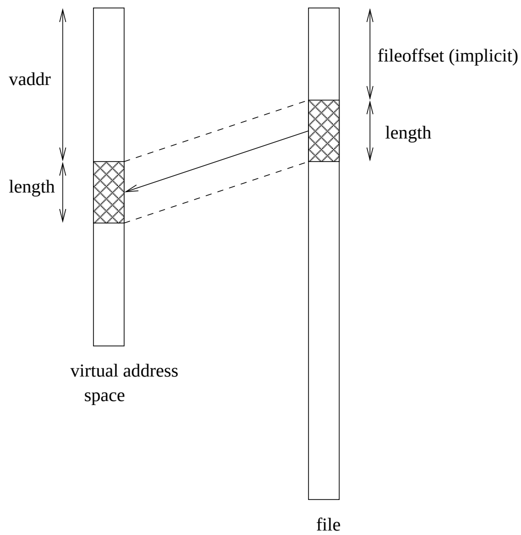
Files and File Systems

- files: persistent, named data objects
  - data consists of a sequence of numbered bytes
  - file may change size over time
  - file has associated meta-data
    * examples: owner, access controls, file type, creation and access timestamps
- file system: a collection of files which share a common name space
  - allows files to be created, destroyed, renamed, ...

File Interface

- open, close
  - open returns a file identifier (or handle or descriptor), which is used in subsequent operations to identify the file. (Why is this done?)
- read, write, seek
  - read copies data from a file into a virtual address space
  - write copies data from a virtual address space into a file
  - seek enables non-sequential reading/writing
- get/set file meta-data, e.g., Unix `fstat, chmod`

File Read



`read(fileID, vaddr, length)`

File Position

- each file descriptor (open file) has an associated file position
- read and write operations
  - start from the current file position
  - update the current file position
- this makes sequential file I/O easy for an application to request
- for non-sequential (random) file I/O, use:
  - a seek operation (`lseek`) to adjust file position before reading or writing
  - a positioned read or write operation, e.g., Unix `pread, pwrite`:
    `pread(fileId,vaddr,length,filePosition)`

Sequential File Reading Example (Unix)

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=0; i<100; i++) {
  read(f,(void *)buf,512);
}
close(f);
```

Read the first $100 * 512$ bytes of a file, 512 bytes at a time.

File Reading Example Using Seek (Unix)

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=1; i<=100; i++) {
  lseek(f,(100-i)*512,SEEK_SET);
  read(f,(void *)buf,512);
}
close(f);
```

Read the first $100 * 512$ bytes of a file, 512 bytes at a time, in reverse order.

File Reading Example Using Positioned Read

```
char buf[512];
int i;
int f = open("myfile",O_RDONLY);
for(i=0; i<100; i+=2) {
  pread(f,(void *)buf,512,i*512);
}
close(f);
```

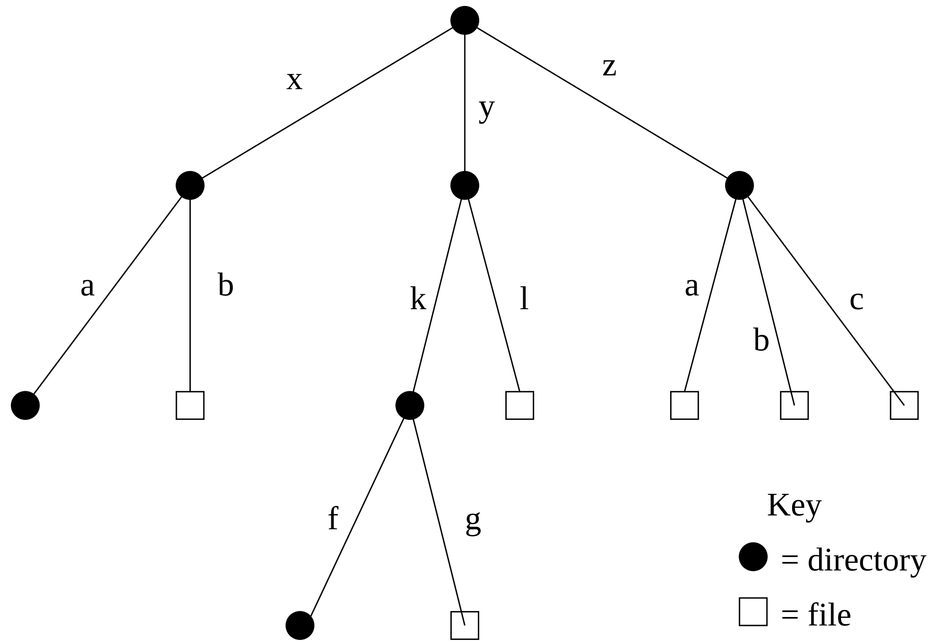
Read every second 512 byte chunk of a file, until 50 have been read.

Directories and File Names

- A directory maps *file names* (strings) to *i-numbers*
  - an i-number is a unique (within a file system) identifier for a file or directory
  - given an i-number, the file system can file the data and meta-data the file
- Directories provide a way for applications to group related files
- Since directories can be nested, a filesystem's directories can be viewed as a tree, with a single *root* directory.
- In a directory tree, files are leaves
- Files may be identified by *pathnames*, which describe a path through the directory tree from the root directory to the file, e.g.:

  `/home/user/courses/cs350/notes/filesys.pdf`
- Directories also have pathnames
- Applications refer to files using pathnames, not i-numbers

Hierarchical Namespace Example



Hard Links

- a *hard link* is an association between a name (string) and an i-number
  - each entry in a directory is a hard link
- when a file is created, so is a hard link to that file
  - `open(/a/b/c,O_CREAT|O_TRUNC)`
  - this creates a new file if a file called `/a/b/c` does not already exist
  - it also creates a hard link to the file in the directory `/a/b`
- Once a file is created, *additional* hard links can be made to it.
  - example: `link(/x/b,/y/k/h)` creates a new hard link `h` in directory `/y/k`. The link refers to the i-number of file `/x/b`, which must exist.
- linking to an existing file creates a new pathname for that file
  - each file has a unique i-number, but may have multiple pathnames
- Not possible to `link` to a directory (to avoid cycles)

Unlinking and Referential Integrity

- hard links can be removed:
  - `unlink(/x/b)`
- the file system ensures that hard links have *referential integrity*, which means that if the link exists, the file that it refers to also exists.
  - When a hard link is created, it refers to an existing file.
  - There is no system call to delete a file. Instead, a file is deleted when its last hard link is removed.

Symbolic Links

- a *symbolic link*, or *soft link*, is an association between a name (string) and a pathname.
  - `symlink(/z/a,/y/k/m)` creates a symbolic link `m` in directory `/y/k`. The symbolic link refers to the pathname `/z/a`.
- If an application attempts to open `/y/k/m`, the file system will
  1. recognize `/y/k/m` as a symbolic link, and
  2. attempt to open `/z/a` instead
- referential integrity is *not* preserved for symbolic links
  - in the example above, `/z/a` need not exist!

UNIX/Linux Link Example (1 of 3)

```
% cat > file1
This is file1.
<cntl-d>
% ls -li
685844 -rw------- 1 user group 15 2008-08-20 file1
% ln file1 link1
% ln -s file1 sym1
% ln not-here link2
ln: not-here: No such file or directory
% ln -s not-here sym2
```

Files, hard links, and soft/symbolic links.

UNIX/Linux Link Example (2 of 3)

```
% ls -li
685844 -rw------- 2 user group 15 2008-08-20 file1
685844 -rw------- 2 user group 15 2008-08-20 link1
685845 lrwxrwxrwx 1 user group  5 2008-08-20 sym1 -> file1
685846 lrwxrwxrwx 1 user group  8 2008-08-20 sym2 -> not-here
% cat file1
This is file1.
% cat link1
This is file1.
% cat sym1
This is file1.
% cat sym2
cat: sym2: No such file or directory
% /bin/rm file1
```

Accessing and manipulating files, hard links, and soft/symbolic links.

UNIX/Linux Link Example (3 of 3)

```
% ls -li
685844 -rw------- 1 user group 15 2008-08-20 link1
685845 lrwxrwxrwx 1 user group  5 2008-08-20 sym1 -> file1
685846 lrwxrwxrwx 1 user group  8 2008-08-20 sym2 -> not-here
% cat link1
This is file1.
% cat sym1
cat: sym1: No such file or directory
% cat > file1
This is a brand new file1.
<cntl-d>
% ls -li
685847 -rw------- 1 user group 27 2008-08-20 file1
685844 -rw------- 1 user group 15 2008-08-20 link1
685845 lrwxrwxrwx 1 user group  5 2008-08-20 sym1 -> file1
685846 lrwxrwxrwx 1 user group  8 2008-08-20 sym2 -> not-here
% cat link1
This is file1.
% cat sym1
This is a brand new file1.
```

Different behaviour for hard links and soft/symbolic links.

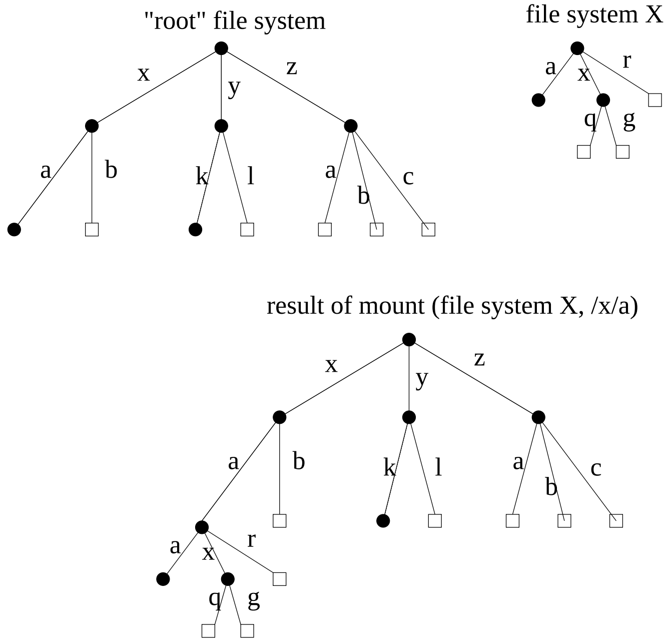
Multiple File Systems

- it is not uncommon for a system to have multiple file systems
- some kind of global file namespace is required
- two examples:

  DOS/Windows: use two-part file names: file system name, pathname within file system
  - example: `C:\user\cs350\schedule.txt`

  Unix: create single hierarchical namespace that combines the namespaces of two file systems
  - Unix `mount` system call does this
- mounting does *not* make two file systems into one file system
  - it merely creates a single, hierarchical namespace that combines the namespaces of two file systems
  - the new namespace is temporary - it exists only until the file system is unmounted

Unix `mount` Example



Links and Multiple File Systems

- hard links cannot cross file system boundaries
  - each hard link maps a name to an i-number, which is unique only *within* a file system
- for example, even after the mount operation illustrated on the previous slide, `link(/x/a/x/g,/z/d)` would result in an error, because the new link, which is in the root file system refers to an object in file system X
- soft links do not have this limitation
- for example, after the mount operation illustrated on the previous slide:
  - `symlink(/x/a/x/g,/z/d)` would succeed
  - `open(/z/d)` would succeed, with the effect of opening `/z/a/x/g`.
- even if the `symlink` operation were to occur *before* the `mount` command, it would succeed

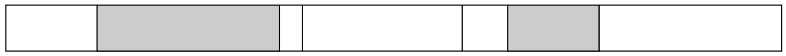
File System Implementation

- what needs to be stored persistently?
  - file data
  - file meta-data
  - directories and links
  - file system meta-data
- non-persistent information
  - open files per process
  - file position for each open file
  - *cached* copies of persistent data

Space Allocation and Layout

- space on secondary storage may be allocated in fixed-size chunks or in chunks of varying size
- fixed-size chunks: *blocks*
  - simple space management
  - internal fragmentation (unused space in allocated blocks)
- variable-size chunks: *extents*
  - more complex space management
  - external fragmentation (wasted unallocated space)

fixed−size allocation



variable−size allocation

Layout matters on secondary storage! Try to lay a file out sequentially, or in large sequential extents that can be read and written efficiently.

File Indexing

- where is the data for a given file?
- common solution: per-file indexing
  - for each file, an index with pointers to data blocks or extents
    - in extent-based systems, need pointer and length for each extent
- how big should the index be?
  - need to accommodate both small files and very large files
  - approach: allow different index sizes for different files

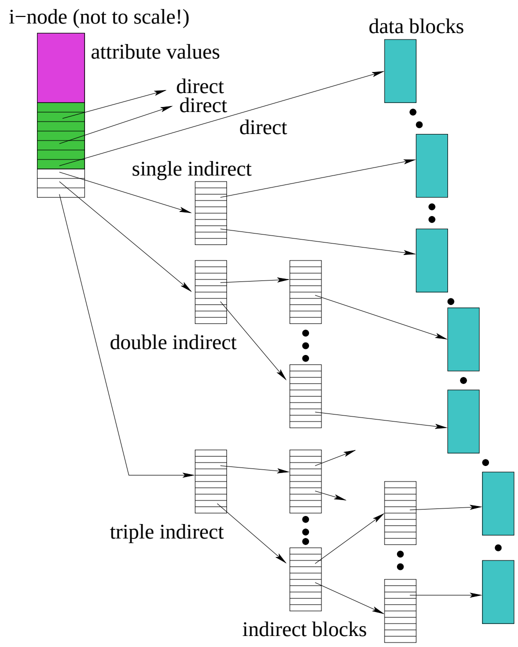
i-nodes

- per file index structure, *fixed size*
- holds file meta-data, and small number of pointers to data blocks
  - for small files, pointers in the i-node are sufficient to point to all data blocks
  - for larger files, allocate additional *indirect* blocks, which hold pointers to additional data blocks
- i-node table holds i-nodes for all files in a file system
  - in persistent storage
  - given i-number, can directly determine location of corresponding i-node in the i-node table

Example: Linux ext3 i-nodes

- i-node fields
  - file type
  - file permissions
  - file length
  - number of file blocks
  - time of last file access
  - time of last i-node update, last file update
  - number of hard links to this file
  - 12 *direct* data block pointers
  - one single, one double, one triple *indirect* data block pointer
- i-node size: 128 bytes
- i-node table: broken into smaller tables, each in a known location on the secondary storage device (disk)

i-node Diagram



Directories

- Implemented as a special type of file.
- Directory file contains directory entries, each consisting of
  - a file name (component of a path name)
  - the corresponding i-number
- Directory files can be read by application programs (e.g., `ls`)
- Directory files are only updated by the kernel, in response to file system operations, e.g, create file, create link
- Application programs cannot write directly to directory files. (Why not?)

Implementing Hard Links

- hard links are simply directory entries
- for example, consider:

  `link(/y/k/g,/z/m)`

- to implement this:
  1. find out the internal file identifier for `/y/k/g`
  2. create a new entry in directory `/z`
     - file name in new entry is `m`
     - file identifier (i-number) in the new entry is the one discovered in step 1

Implementing Soft Links

- soft links can be implemented as a special type of file
- for example, consider:

  `symlink(/y/k/g,/z/m)`

- to implement this:
  - create a new *symlink* file
  - add a new entry in directory `/z`
    * file name in new entry is `m`
    * i-number in the new entry is the i-number of the new symlink file
  - store the pathname string “/y/k/g” as the contents of the new symlink file

Pathname Translation

- input: a file pathname
- output: the i-number of the file the pathname refers to
- common to many file system calls, e.g., open
- basic idea (without error checking):

```
i = i-number of root directory
while (n = next component of pathname) {
   if i is not a directory then return ERROR
   i = lookup n in directory i
   if (i is a symbolic link file) {
      i = translate(link)
   }
}
return i
```


In-Memory (Non-Persistent) Structures

- per process
  - descriptor table
    * which file descriptors does this process have open?
    * to which file does each open descriptor refer?
    * what is the current file position for each descriptor?
- system wide
  - open file table
    * which files are currently open (by any process)?
  - i-node cache
    * in-memory copies of recently-used i-nodes
  - block cache
    * in-memory copies of data blocks and indirect blocks

Problems Caused by Failures

- a single logical file system operation may require several disk I/O operations
- example: deleting a file
  - remove entry from directory
  - remove file index (i-node) from i-node table
  - mark file's data blocks free in free space index
- what if, because of a failure, some but not all of these changes are reflected on the disk?

- system failure will destroy in-memory file system structures
- persistent structures should be *crash consistent*, i.e., should be consistent when system restarts after a failure

---

Fault Tolerance

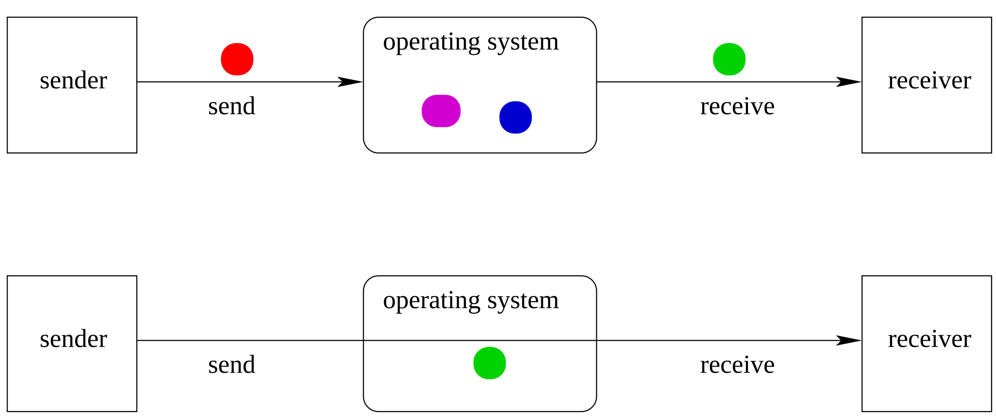
- special-purpose consistency checkers (e.g., Unix `fsck` in Berkeley FFS, Linux ext2)
  - runs after a crash, before normal operations resume
  - find and attempt to repair inconsistent file system data structures, e.g.:
    - file with no directory entry
    - free space that is not marked as free
- journaling (e.g., Veritas, NTFS, Linux ext3)
  - record file system meta-data changes in a journal (log), so that sequences of changes can be written to disk in a single operation
  - *after* changes have been journaled, update the disk data structures (*write-ahead logging*)
  - after a failure, redo journaled updates in case they were not done before the failure

Interprocess Communication Mechanisms

- shared storage
  - shared virtual memory
  - shared files
- message-based
  - signals
  - sockets
  - pipes
  - . . .

Message Passing

Indirect Message Passing



Direct Message Passing

If message passing is indirect, the message passing system must have some capacity to buffer (store) messages.

Properties of Message Passing Mechanisms

Directionality:

- simplex (one-way), duplex (two-way)
- half-duplex (two-way, but only one way at a time)

Message Boundaries:

datagram model: message boundaries

stream model: no boundaries

Connections: need to connect before communicating?

- in connection-oriented models, recipient is specified at time of connection, not by individual send operations. All messages sent over a connection have the same recipient.
- in connectionless models, recipient is specified as a parameter to each send operation.

Reliability:

- can messages get lost? reordered? damaged?

Sockets

- a socket is a communication *end-point*
- if two processes are to communicate, each process must create its own socket
- two common types of sockets

  stream sockets: support connection-oriented, reliable, duplex communication under the stream model (no message boundaries)

  datagram sockets: support connectionless, best-effort (unreliable), duplex communication under the datagram model (message boundaries)

- both types of sockets also support a variety of address domains, e.g.,

  Unix domain: useful for communication between processes running on the same machine

  INET domain: useful for communication between process running on different machines that can communicate using IP protocols.

Using Datagram Sockets (Receiver)

```
s = socket(addressType, SOCK_DGRAM);
bind(s,address);
recvfrom(s,buf,bufLength,sourceAddress);
...
close(s);
```

- `socket` creates a socket
- `bind` assigns an address to the socket
- `recvfrom` receives a message from the socket
  - `buf` is a buffer to hold the incoming message
  - `sourceAddress` is a buffer to hold the address of the message sender
- both `buf` and `sourceAddress` are filled by the `recvfrom` call

Using Datagram Sockets (Sender)

```
s = socket(addressType, SOCK_DGRAM);
sendto(s,buf,msgLength,targetAddress)
...
close(s);
```

- `socket` creates a socket
- `sendto` sends a message using the socket
  - `buf` is a buffer that contains the message to be sent
  - `msgLength` indicates the length of the message in the buffer
  - `targetAddress` is the address of the socket to which the message is to be delivered

More on Datagram Sockets

- `sendto` and `recvfrom` calls *may* block
  - `recvfrom` blocks if there are no messages to be received from the specified socket
  - `sendto` blocks if the system has no more room to buffer undelivered messages
- datagram socket communications are (in general) unreliable
  - messages (datagrams) may be lost
  - messages may be reordered
- The sending process must know the address of the receive process's socket.

Using Stream Sockets (Passive Process)

```
s = socket(addressType, SOCK_STREAM);
bind(s,address);
listen(s,backlog);
ns = accept(s,sourceAddress);
recv(ns,buf,bufLength);
send(ns,buf,bufLength);
...
close(ns); // close accepted connection
close(s);  // don't accept more connections
```

- `listen` specifies the number of connection requests for this socket that will be queued by the kernel
- `accept` accepts a connection request and creates a new socket (`ns`)
- `recv` receives up to `bufLength` bytes of data from the connection
- `send` sends `bufLength` bytes of data over the connection.

Notes on Using Stream Sockets (Passive Process)

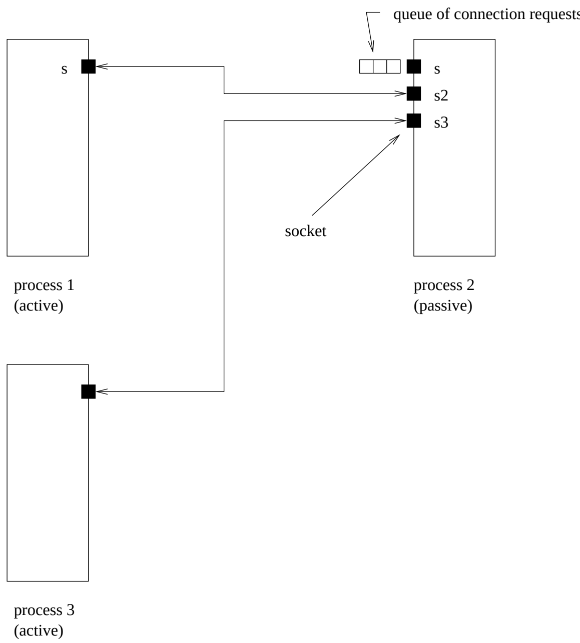
- `accept` creates a new socket (`ns`) for the new connection
- `sourceAddress` is an address buffer. `accept` fills it with the address of the socket that has made the connection request
- additional connection requests can be accepted using more `accept` calls on the original socket (`s`)
- `accept` blocks if there are no pending connection requests
- connection is duplex (both `send` and `recv` can be used)

Using Stream Sockets (Active Process)

```
s = socket(addressType, SOCK_STREAM);
connect(s,targetAddress);
send(s,buf,bufLength);
recv(s,buf,bufLength);
...
close(s);
```

- `connect` sends a connection request to the socket with the specified address
  - `connect` blocks until the connection request has been accepted
- active process may (optionally) bind an address to the socket (using `bind`) before connecting. This is the address that will be returned by the `accept` call in the passive process
- if the active process does not choose an address, the system will choose one

Illustration of Stream Socket Connections



Pipes

- pipes are communication objects (not end-points)
- pipes use the stream model and are connection-oriented and reliable
- some pipes are simplex, some are duplex
- pipes use an implicit addressing mechanism that limits their use to communication between *related* processes, typically a child process and its parent
- a `pipe()` system call creates a pipe and returns two descriptors, one for each end of the pipe
  - for a simplex pipe, one descriptor is for reading, the other is for writing
  - for a duplex pipe, both descriptors can be used for reading and writing

One-way Child/Parent Communication Using a Simplex Pipe

```
int fd[2];
char m[] = "message for parent";
char y[100];
pipe(fd); // create pipe
pid = fork(); // create child process
if (pid == 0) {
  // child executes this
  close(fd[0]); // close read end of pipe
  write(fd[1],m,19);
  ...
} else {
  // parent executes this
  close(fd[1]); // close write end of pipe
  read(fd[0],y,19);
  ...
}
```


Illustration of Example (after `pipe()`)

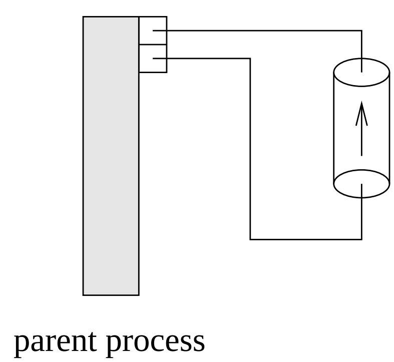


Illustration of Example (after `fork()`)

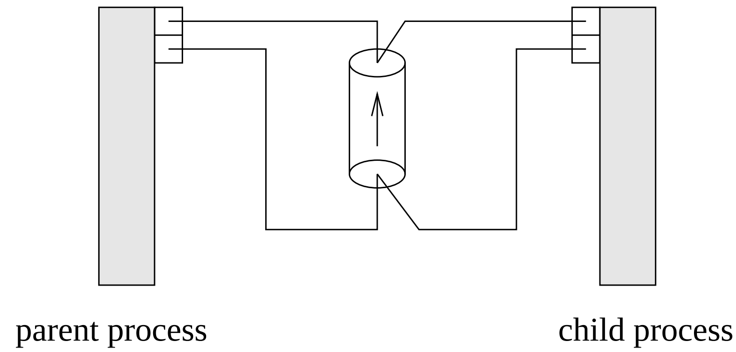
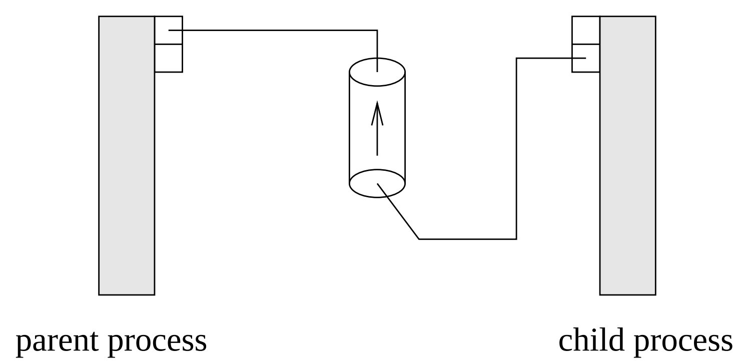
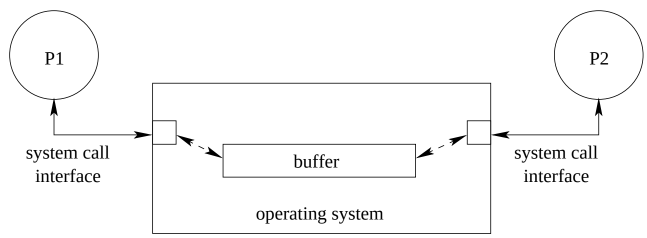


Illustration of Example (after `close()`)



Implementing IPC

- application processes use descriptors (identifiers) provided by the kernel to refer to specific sockets and pipes, as well as files and other objects
- kernel *descriptor tables* (or other similar mechanism) are used to associate descriptors with kernel data structures that implement IPC objects
- kernel provides bounded buffer space for data that has been sent using an IPC mechanism, but that has not yet been received
  - for IPC objects, like pipes, buffering is usually on a per object basis
  - IPC end points, like sockets, buffering is associated with each endpoint



Network Interprocess Communication

- some sockets can be used to connect processes that are running on different machines
- the kernel:
  - controls access to network interfaces
  - multiplexes socket connections across the network