

Reading Reference:

Textbook: Chapter 7

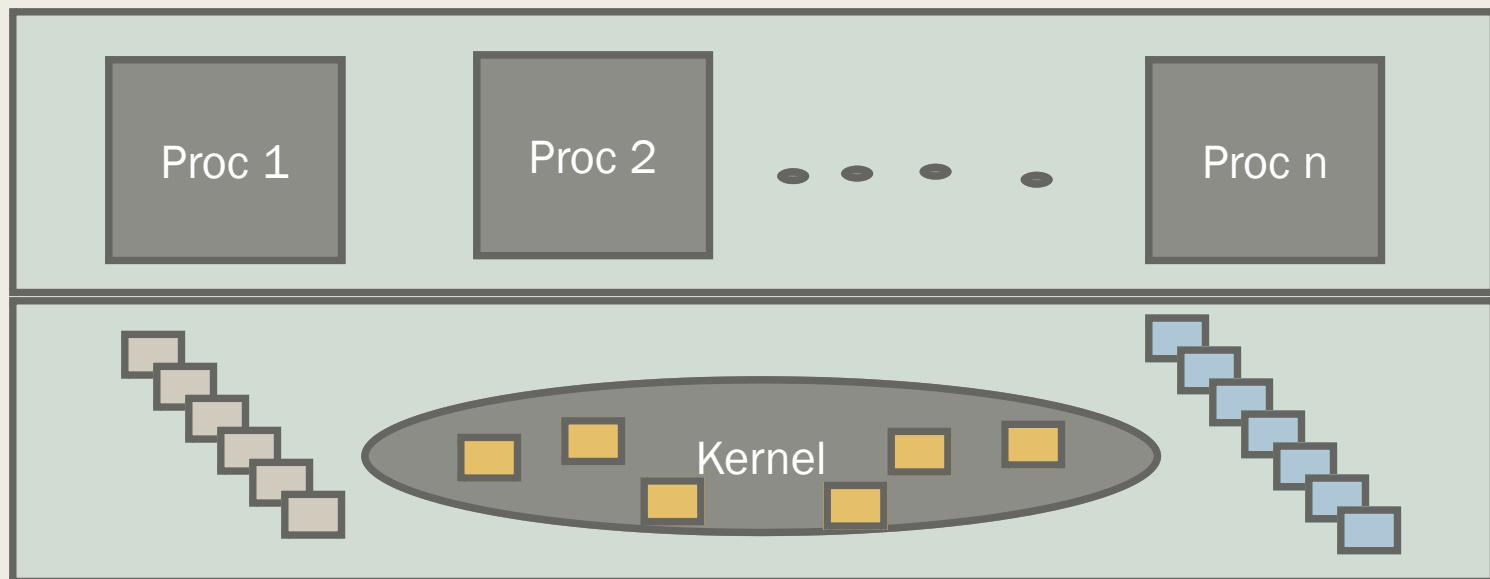
# UNIX PROCESS SCHEDULING

Tanzir Ahmed

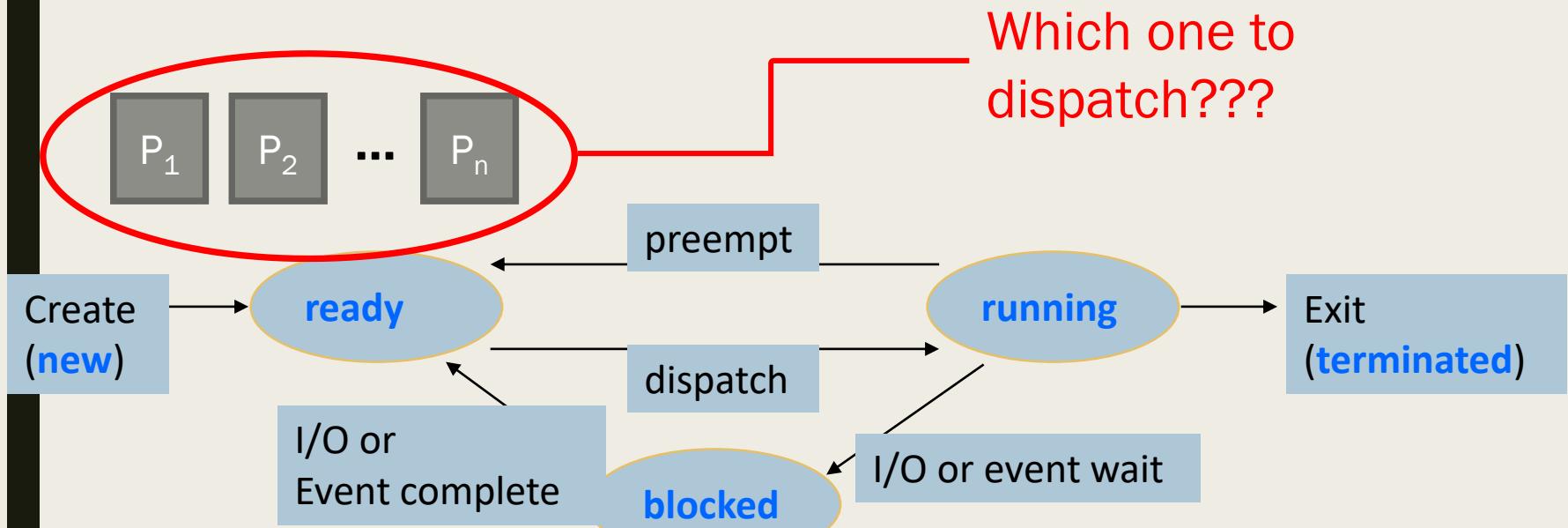
CSCE 313 Spring 2020

# Process Scheduling

- Today we will ask how does a Kernel juggle the (often) competing requirements of **Performance**, **Fairness**, **Utilization**, etc. in dealing with concurrency

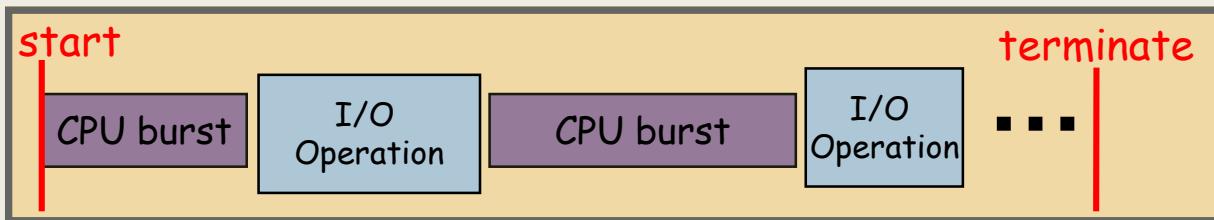


# Process Scheduling – More Specifically



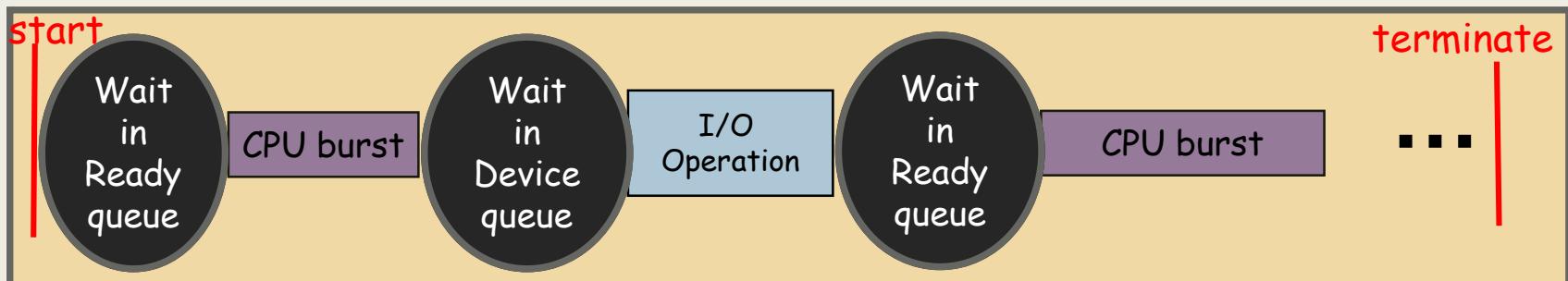
# Overall Idea

If there were no other processes, a process's life would probably look like the following:



However, since there are other processes, each process is slowed down by waits in Ready and Blocked queues:

- *Think of CPU and I/O devices as servers in the bank, where people must wait in line to get service*



Now, the scheduler's goal is to minimize these Wait time in some aggregate sense, especially in the Ready queue

- *Our main focus is CPU scheduler*
- *There are other schedulers (e.g., I/O schedulers) are not our focus*

# Scheduling Metrics

- Task/Job
  - *User request: e.g., mouse click, web request, shell command, ...*
- Latency/Response Time
  - *How long does a task take to complete?*
- Throughput
  - *How many tasks can be done per unit of time?*
- Overhead
  - *How much extra work is done by the scheduler?*
- Fairness
  - *How equal is the performance received by different users?*
- Predictability
  - *How consistent is the performance over time?*

# How to Measure?

- **Waiting Time:** Time spent in Ready queue
  - *IOW, Time between job's arrival in the Ready (or Blocked) queue and start of service*
- **Service (Execution) Time:** Time spent in CPU (or I/O device)
- **Response (Completion) Time:**  
*Response Time = Finish Time - Arrival Time*
  - Response time is what the user experiences:
    - Time to echo a keystroke in editor
    - Time to compile a program
  - Another view of Resp Time is the time it waits in line, plus the time it is processed:

$$\text{Response Time} = \text{Waiting Time} + \text{Service Time}$$

- **Throughput:** number of jobs completed per unit of time
  - *The more the device utilization, the higher the throughput*
  - *Throughput related to response time, but not same thing:*
    - Minimizing response time and maximizing throughput can be contradictory (examples coming up)

# Scheduling Metrics and Angles

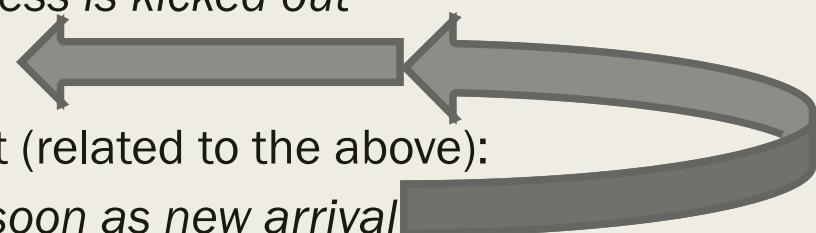
- Scheduling metrics are important for both individual processes and the system as a whole
  - ***Customer-Centric:*** Response Time (i.e., time to finish a given task)
  - ***System-Centric:*** Throughput , Average Response Time, Fairness

# Scheduling Goals

- 1. **Minimize** Average Response Time (ART)
  - *Measure from the response time of a number of jobs*
  - *Also applies to 1 job as – i.e., we want to minimize the response time for processing a mouse click, a file copy, to compile a program etc.*
- 2. **Maximize** Throughput
  - *By keeping CPU and I/O devices as utilized as possible*
- 3. **Ensure** Fairness
  - *Share CPU in some equitable way*
  - *This policy can contradict with others (we will see examples)*

# Scheduling: Decision Points

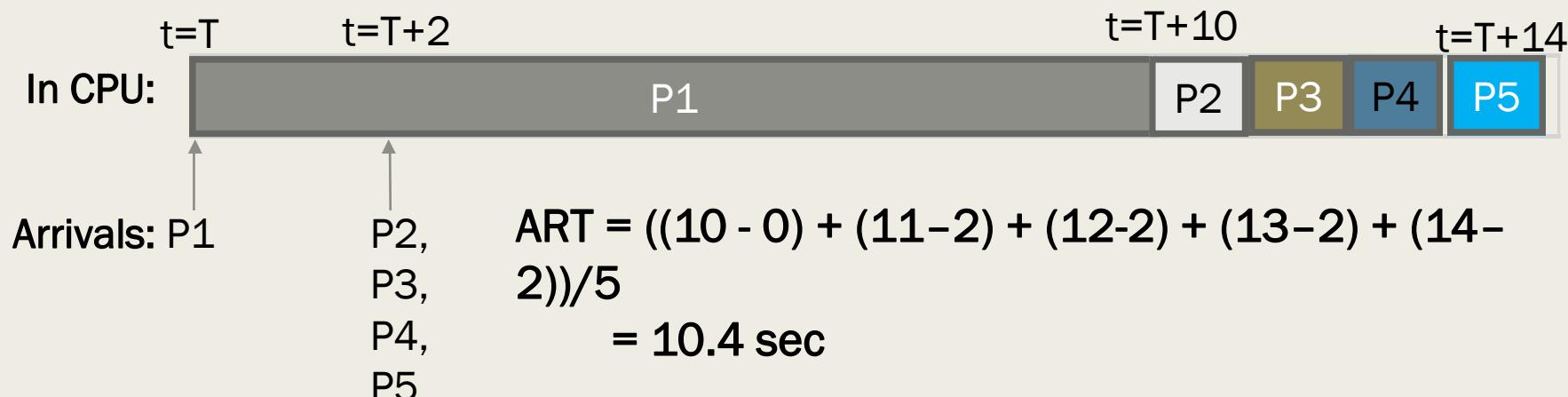
- There can be 3 points in time when the scheduler runs:
  - *The current process voluntarily gives up (by requesting I/o) CPU*
  - *Timer expired - the current process is kicked out*
  - ***Arrival of a new process (new!!)***
- A scheduler can be preemptive or not (related to the above):
  - ***Preemptive:*** scheduler runs as soon as new arrival
  - ***Non-preemptive:*** scheduler waits until timer expires, or the current process yields CPU
- Some scheduling algorithms can run in both preemptive and non-preemptive mode
  - *SRTF can be run in both preemptive or non-preemptive manner. FIFO would be the same with or w/o preemption*
  - *Round Robin is non-preemptive by definition*



# P1: First In First Out (FIFO) or FCFS (First Come First Served)

- Schedule tasks in the order they arrive
  - Continue running them until they complete or give up the processor*
- Uses:
  - Read/write requests to a disk file*
  - Network packets in a NIC*
- An Example:

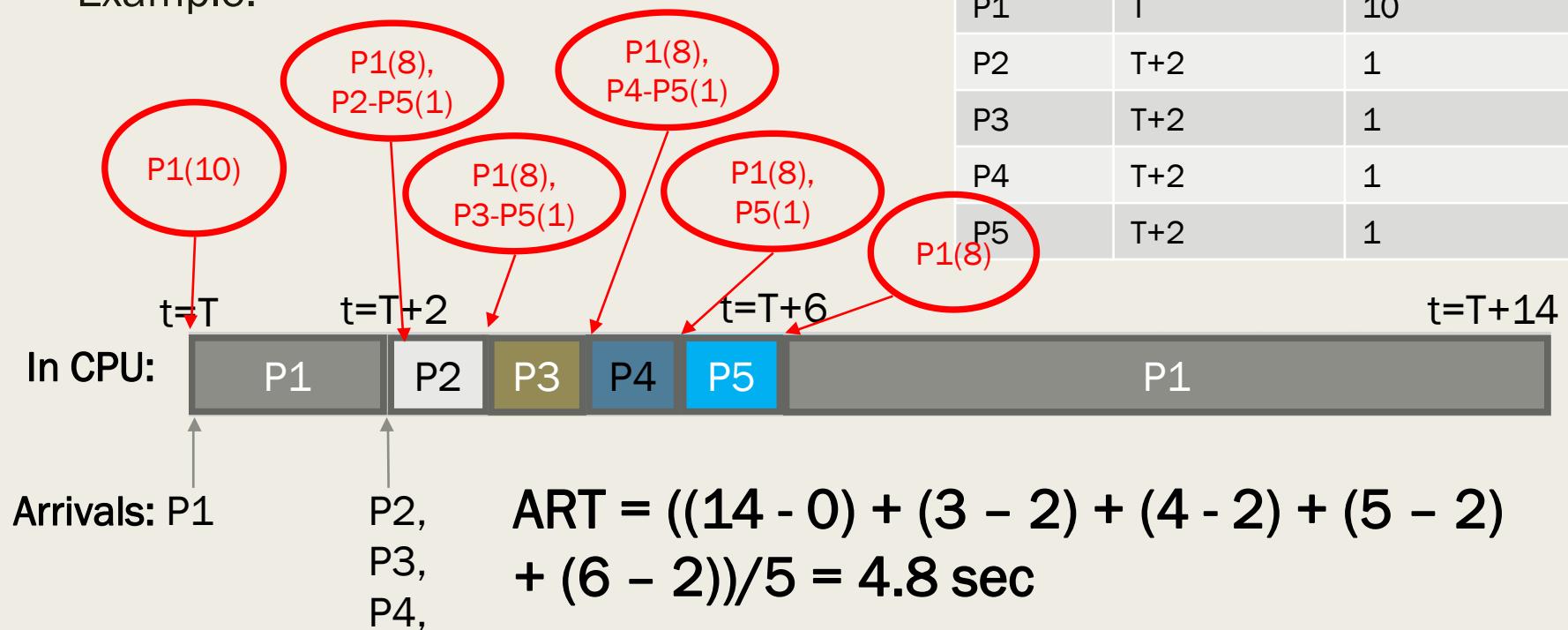
Job	Arrival Time (sec)	Service Time (sec)
P1	T	10
P2	T+2	1
P3	T+2	1
P4	T+2	1
P5	T+2	1



# P2: Shortest Remaining Time First (SRTF)

- Always do the task that has the shortest remaining amount of work to do
  - Aka. *Shortest Job First (SJF)*
  - Note: *Remaining time of a job changes every time it gets service in CPU*

Example:



# Some Thoughts – Why the Reality has Hope

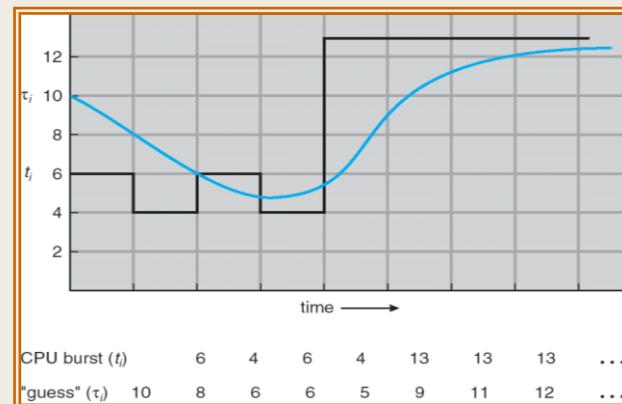
- Based on which queue a process spends most of its time, we can classify processes to 2 types:
- I/O-bound Processes
  - Spend most time in I/O queues
  - Example: A web-server reading requests from network (I/O), reading files from disk (I/O), writing response back to network (I/O), and some request parsing (CPU)
- CPU-bound Processes
  - Spend most time in ready queue or as running
  - Example: A program computing large prime numbers – a lot of computations involving CPU, but no/less I/O

# Some Thoughts

- Claim: SRTF is optimal for ART
  - *SRTF always picks the shortest job; if it did not, then, by definition, it would result in higher average response time.* <<see notes for details>>
- When is FIFO optimal?
  - *When all jobs are equal in length*
  - *SRTF gives the same schedule, but incurs many context switches*
- Does SRTF have any downsides?
  - *Starvation: longer jobs would suffer. Imagine a supermarket that implements SRTF in checkout lines! Longer lines will keep waiting*
  - *Implementation: No exact algorithm to implement SRTF (how would you know how much is remaining???)*

# Practical Implementation of SRTF

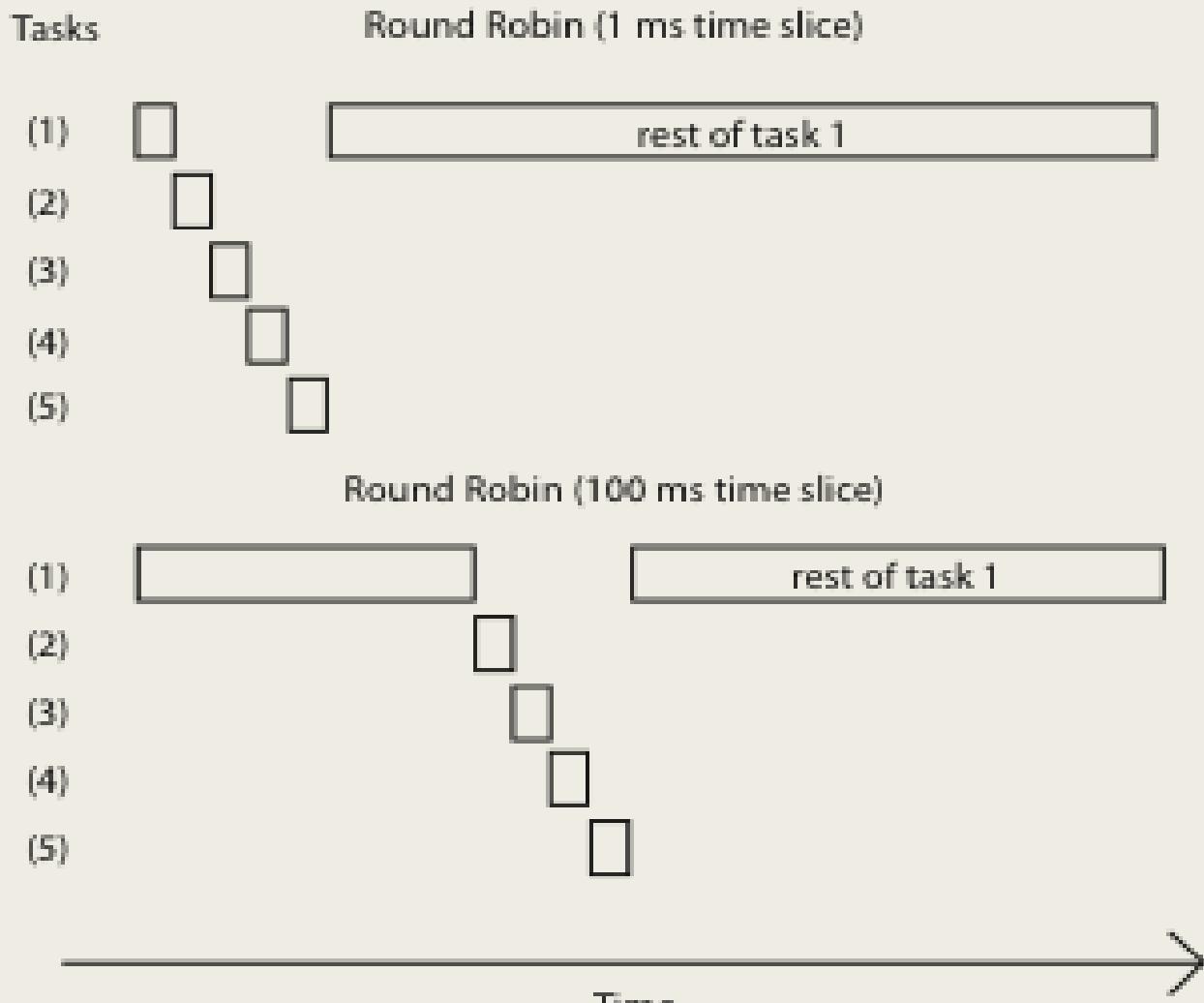
- Issue: How do we know the remaining time?
  - *User provides job runtime*
    - System kills job if takes too long (i.e., to stop cheating)
  - *But even for non-malicious users, it is hard to predict runtime accurately*
- Adaptive Algorithm without user input: Predict the **next CPU burst** (not the whole task length) based on the recent past
  - *Works because programs have predictable behavior*
    - If program was I/O bound in past, probably it will be in future
- Example: SRTF with estimated burst length
  - *Use an estimator function on previous bursts:*  
*Let  $t_{n-1}$ ,  $t_{n-2}$ ,  $t_{n-3}$ , etc. be previous CPU burst lengths.*  
*Estimate next burst  $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \dots)$*
  - *Function  $f$  could be one of many different time series estimation schemes (Kalman filters, etc.)*



# P3: Round Robin

- Each task gets resource for a fixed **time quantum**
  - *If task doesn't complete, it goes back in line*
  - *If it finishes the CPU burst because of I/O, it gets out before the quantum expires*
- Now, we need a timer, right???
  - *So far, we have been operating w/o a timer*
- But, **how to we choose a good time quantum???**
  - *Too long (i.e., Infinite)?*
    - *Then it will be equivalent to FIFO (as if there is no timer and no preemption)*
  - *Too short (i.e. One instruction)?*
    - *Too much overhead of swapping processes*
    - *But tasks finish approximately in the order of their length (approximating SRTF)*

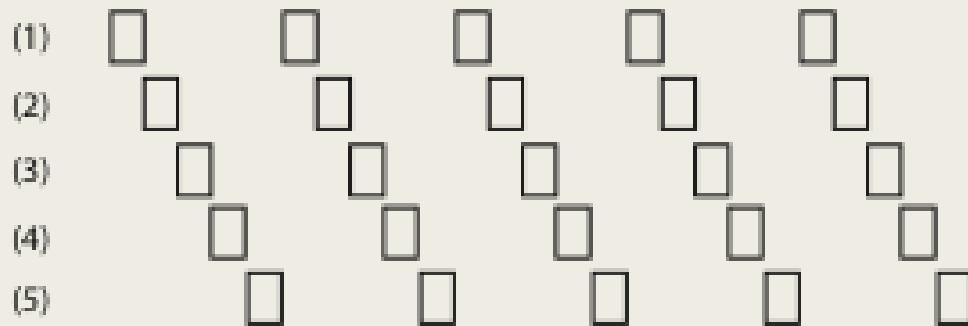
# Round Robin – Varying Time Slice



# Round Robin vs. FIFO

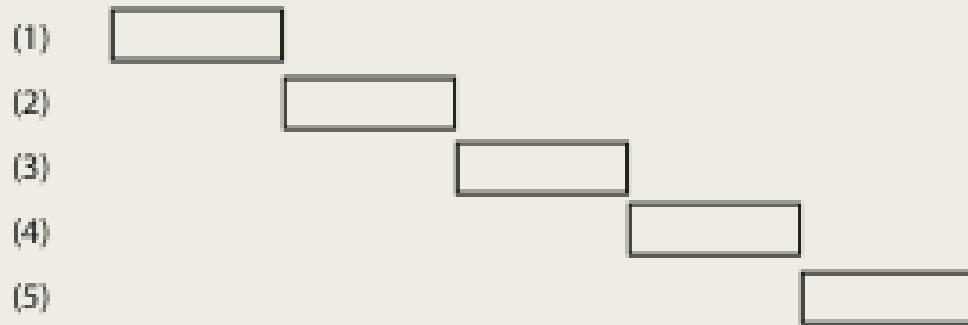
Tasks

Round Robin (1 ms time slice)



Average Response Time  
 $= (21+22+23+24+25)/5$   
 $= 23$

FIFO and SJF



Average Response Time  
 $= (5+10+15+20+25)/5$   
 $= 15$

Time

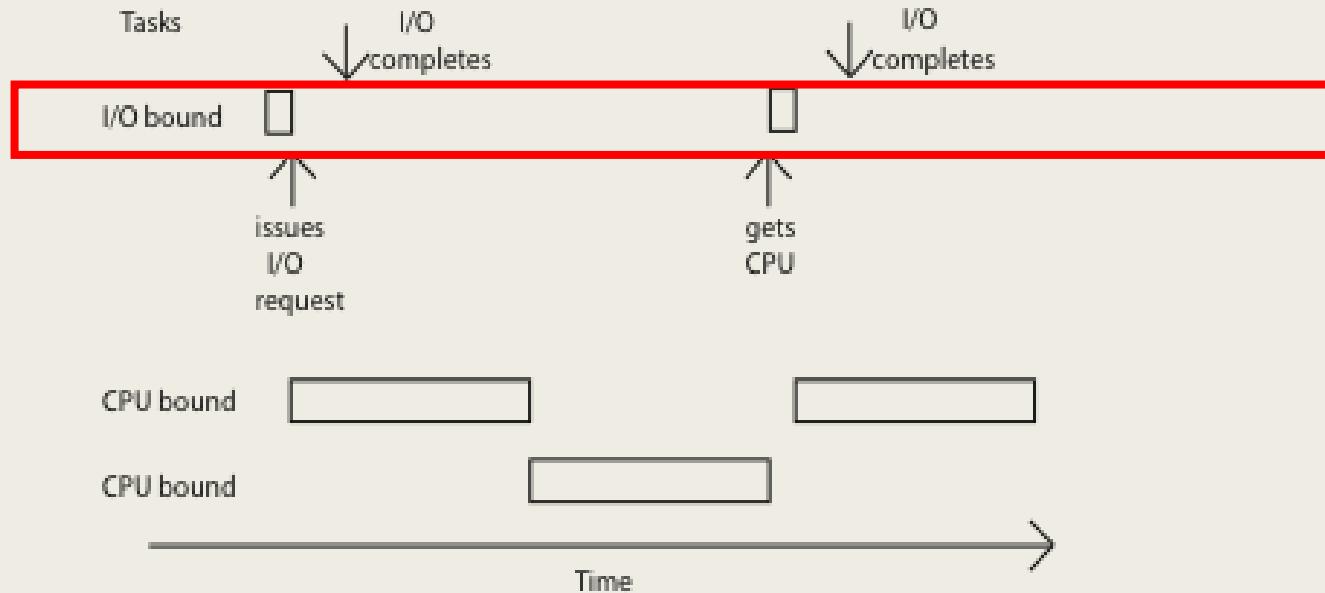
# Round Robin vs. FIFO

- Assuming zero-cost context-switch, is Round Robin always better than FIFO?
  - *No. Round robin is better when there is a mix of short and long jobs. However, it is poor for jobs that are the same length*
  - *Of course, context switches are not zero-cost, adding more overhead to RR*
- What's the worst case for RR?
  - *All jobs are of same length*
  - *All tasks run a factor slower than the best case*
  - *CPU devoted to Context Switch Overhead is without any benefit*

# Round Robin vs. Fairness

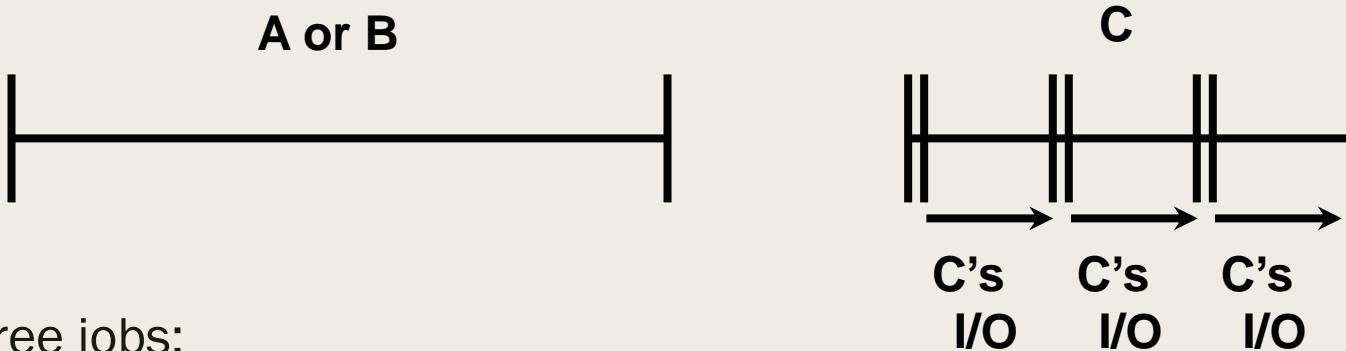
- Is Round Robin always **fair**?
  - *Yes. Round robin ensures nobody starves, and gives everyone a turn*
  - *But lets short tasks complete before long tasks*

# Another Downside of RR – Mixed Workload



- I/O task (e.g., keystroke) must wait for its turn for the CPU
  - Gets a tiny fraction of the performance it could get
- We could shorten the RR quantum - would help, but it would increase overhead
- What would this do under **SRTF**?
  - Every time the task is ready, it is scheduled immediately!

# Example - Benefits of SRTF



Three jobs:

- *A,B: CPU bound, each run for a week*  
*C: I/O bound, loop 1ms CPU, 9ms disk I/O*
- *If only one at a time, C uses 90% of the disk, A or B use 100% of the CPU*

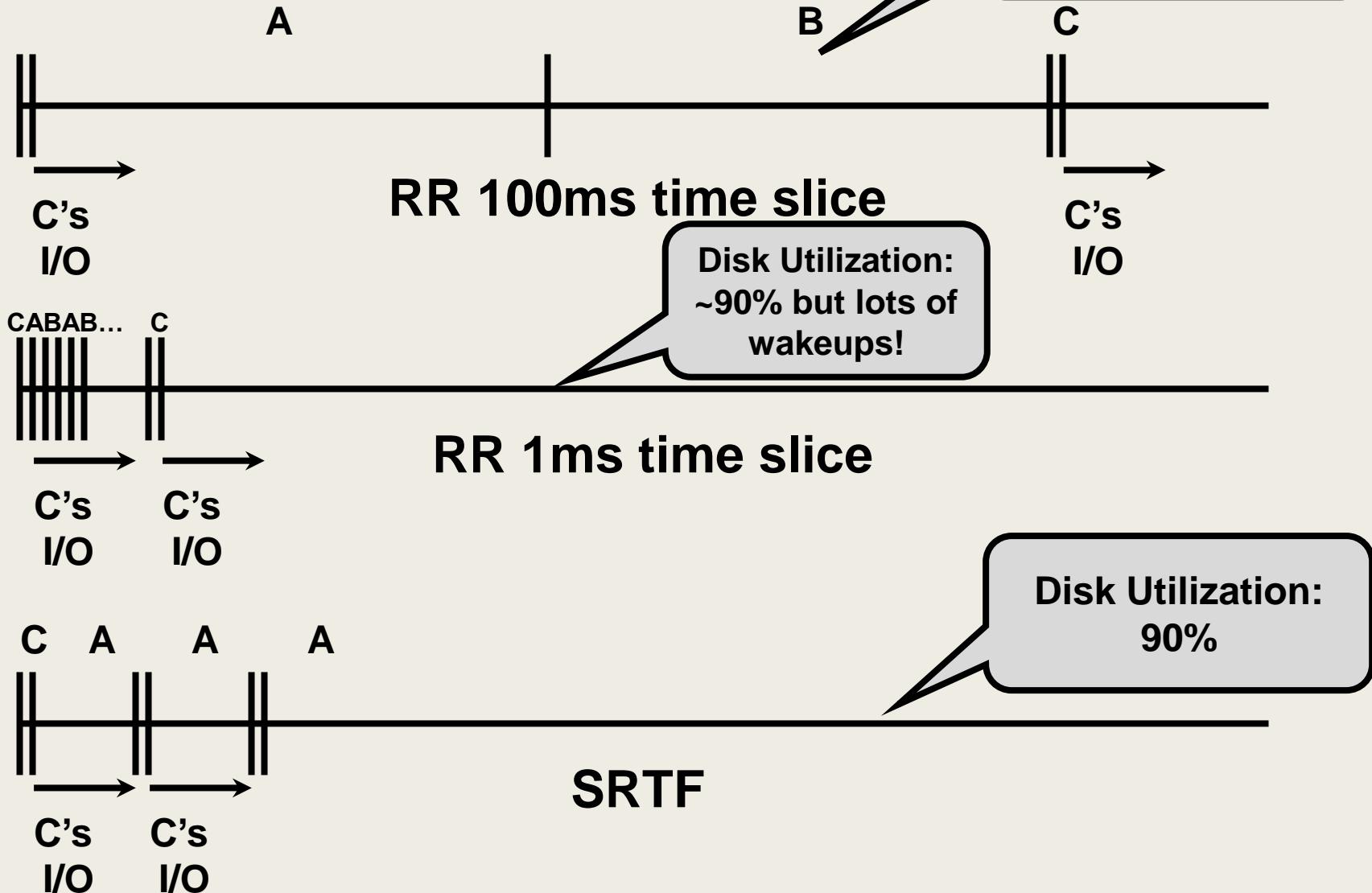
With FIFO:

- *Once A or B get in, keep CPU for one week each*

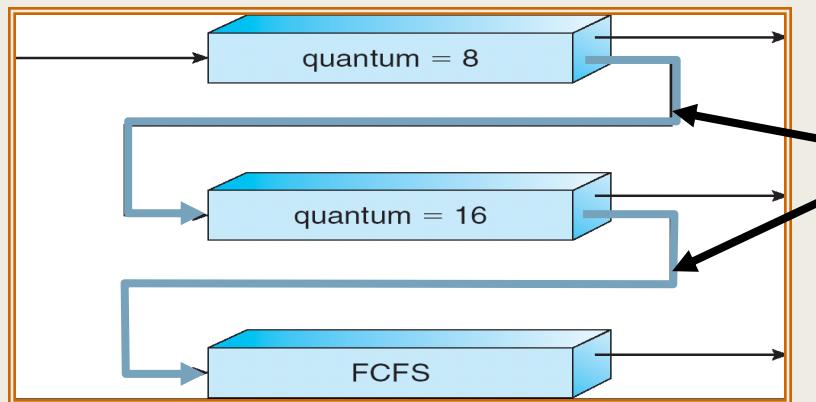
What about RR or SRTF?

- *Easier to see with a timeline*

# RR vs. SRTF



# Multi-Level Feedback Scheduling



- Another method for exploiting past behavior
  - *First used in Cambridge Time Sharing System (CTSS)*
  - *Multiple queues, each with different priority*
    - Higher priority queues often considered “foreground” tasks
  - *Each queue has its own scheduling algorithm*
    - e.g., foreground – RR, background – FCFS
    - Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc.)
- Adjust each job’s priority as follows (details vary)
  - *Job starts in highest priority queue*
  - *If timeout expires, drop one level*

# Countermeasure

- Countermeasure: User action that can foil intent of the OS designer
  - *Put in a bunch of meaningless I/O to keep job's priority high*
  - *Of course, if everyone did this, wouldn't work!*
- Example: MIT Othello game project (simpler version of Go game)
  - *Computer playing against competitor's computer, so key was to do computing at higher priority than the competitors.*
    - Cheater (or Winner!!!) put in printf's, ran much faster!

# MLFQ Scheduling Details

- Result approximates SRTF:
  - *CPU bound jobs drop like a rock*
  - *Short-running I/O bound jobs stay near top*
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    - Serve all from highest priority, then next priority, etc.
    - But this tends to be unfair
    - Example: In Multics, people found a 10-year old job while shutting down
  - Time slice:
    - Each queue gets a certain amount of CPU time
    - e.g., 70% to highest, 20% next, 10% lowest
    - More fair compared to fixed priority, so this one used in practice



# Summary

- FCFS is simple and minimizes overhead.
- If tasks are variable in size, then FCFS can have very poor ART
- If tasks are equal in size, FCFS is optimal in terms of ART
  - *SRTF becomes equivalent to FCFS*
  - *RR would do increase ART significantly*
- SRTF is optimal in terms of ART
  - *The only way to implement is for Kalman filter-like approximations for the individual CPU bursts*
  - *But NOT fair*
  - *We also do not have preemption – longer jobs can have very long waiting times, making them unresponsive*

# Summary (contd.)

- If tasks are equal in size, RR will have poor ART
- RR works poorly on a mix of CPU and I/O bound tasks
  - *SJF is hugely beneficial in this case*
- RR avoids starvation and is fair
- To avoid the downsides of RR and SRTF, we want something in the middle
  - *That would combine the good sides of both*
- MFQ scheduler is the most practical – answer to our prayer
  - *Approximates SJF while running just RR for each queue, no need to run any adaptive algorithm for the CPU bursts*
  - *Achieves a balance between responsiveness, low overhead, and fairness*
  - *Good for mixed workloads (user typing and long CPU computations running simultaneously)*

Reading Reference:

Textbook1: Chapter 4 (Section 4.1-4.4)

# CONCURRENCY AND THREADS

Tanzir Ahmed  
CSCE 313 Spring 2020

# Today's Conversation

## ■ Threads

- *A bit complex topic but central to our understanding of modern computer systems (HW and SW)*
- *We will build concepts incrementally and tie the picture together at the end*

*Adapted from contemporary courses in OS/Systems taught at Berkeley, UW, TAMU, UIUC, and Rice. Special acknowledgment to Profs Gu/Bettati/Tyagi at TAMU, Culler and Joseph at Berkeley*

# Why Processes & Threads?

## Goals:

- **Multiprogramming:** Run multiple applications concurrently
- **Protection:** Don't want a bad application to crash system!

## Solution:

- Virtual Machine abstraction: give process illusion it owns machine (i.e., CPU, Memory, and IO device multiplexing)
- Process: unit of execution and allocation

## Challenge:

- Process creation & switching expensive
- Need concurrency within same app (e.g., web server)

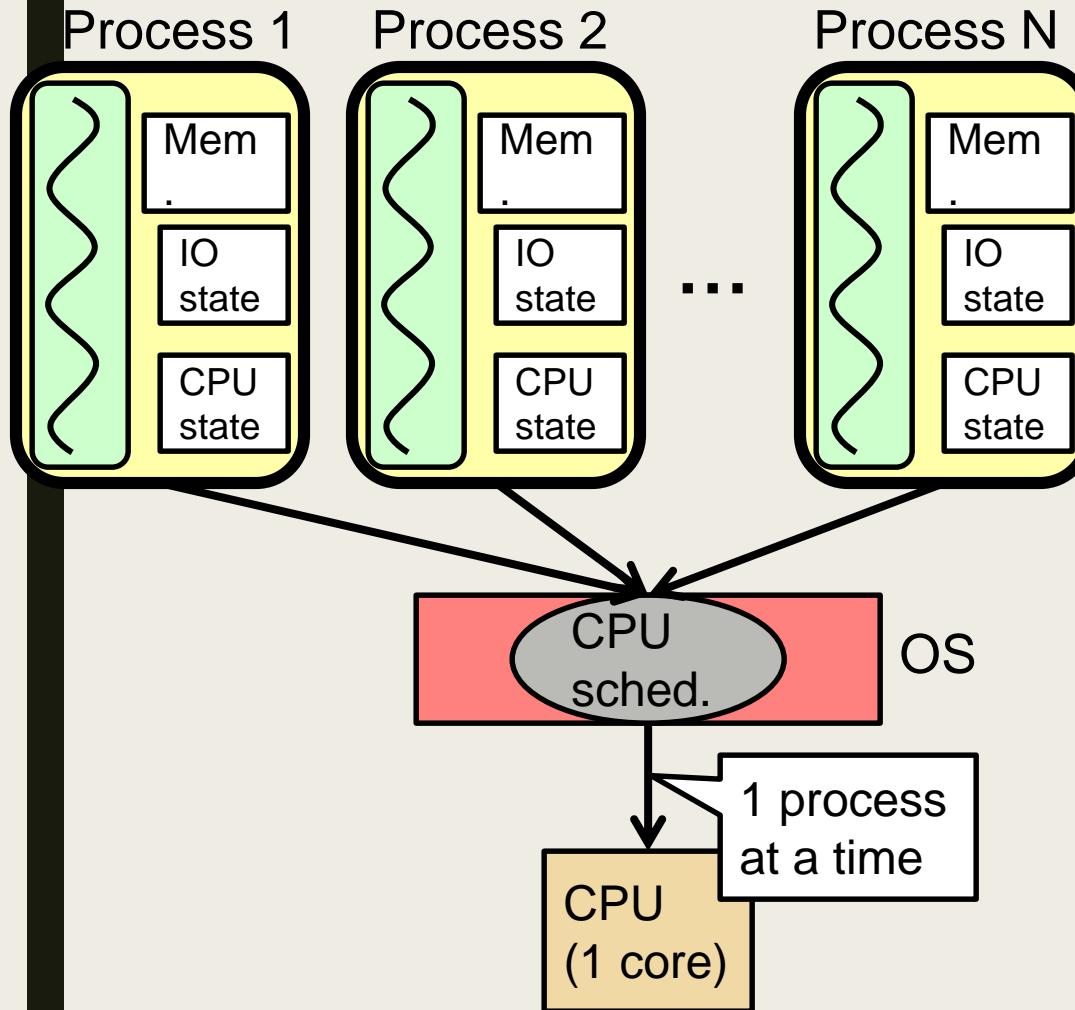
## Solution:

- Thread: Decouple allocation and execution
- Run multiple threads within same process

# Motivation for Threads

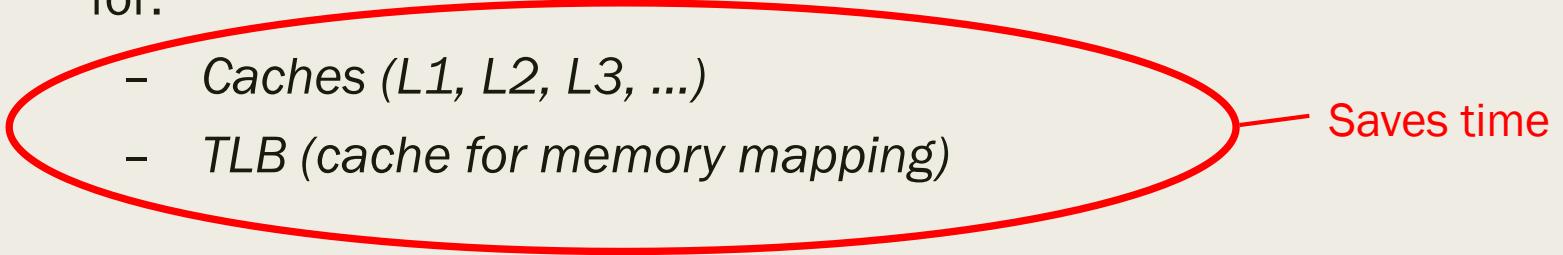
- Process Context Switch has huge overhead. Requires switching:
  - CPU state (*PC, PSW, all registers etc*) - fast
  - Memory Address Space - **SLOW**
- Assumption: memory access without cache is slow
- A Memory Address Space comes with:
  - Virtual Memory System data
    - Page tables (mapping) and Translation Look-aside Buffer (TLB) (i.e., working as cache for mapping)
  - Actual Memory Content
    - Dirty (modified) pages that are updated only in cache, not in memory
    - Need to be updated in the memory before switch
- All of these become obsolete after a switch
  - *All caches (L1, L2, ..), TLB become COLD*
  - *Require time to WARM UP as the new process runs*

# Processes Context Switch



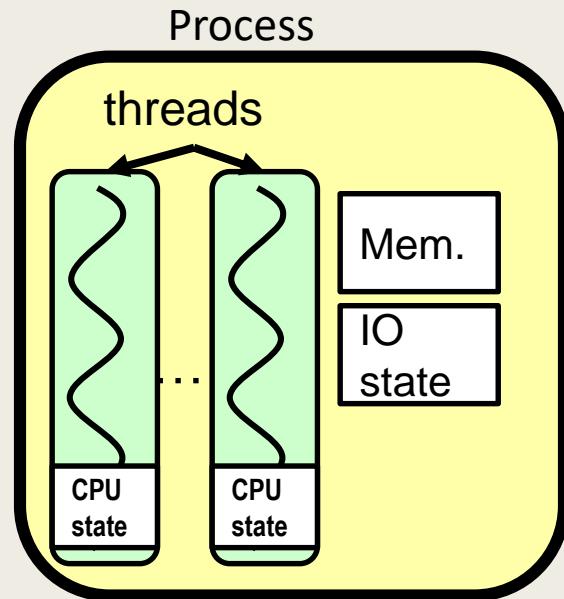
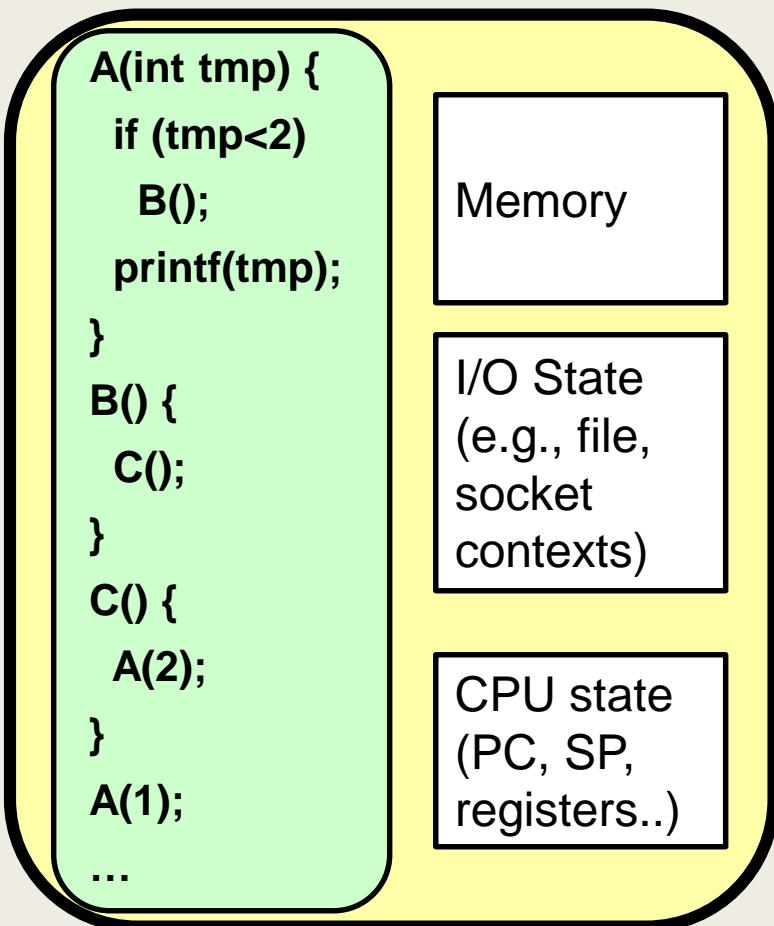
- Process Switch overhead:  
**high**
  - CPU state: *low*
  - Memory/IO state: **high**
- Process creation: **high**
- Protection
  - CPU: **yes**
  - Memory/IO: **yes**
- Sharing overhead: **high**  
(involves at least a context switch)

# Thread Context Switch

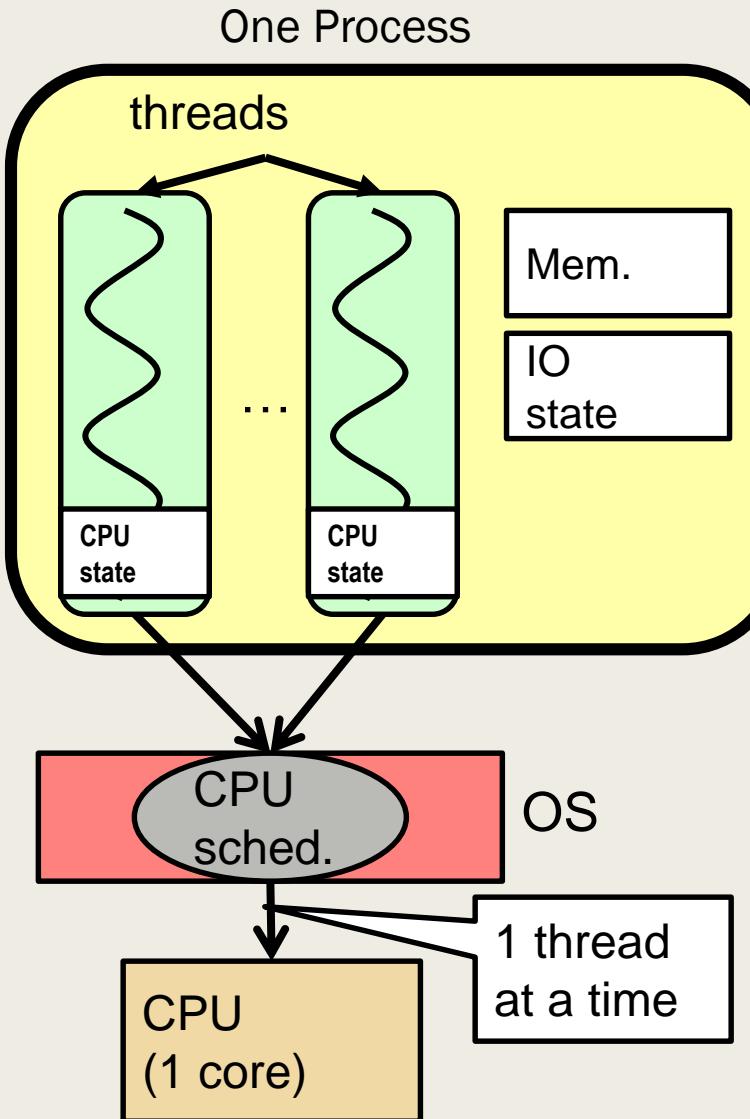
- Still requires:
    - *The kernel scheduler to run, that overhead stays*
    - *Save the current CPU context, and load the next one*
  - But, the old address space stays. Thus, no **WARM UP** required for:
    - Caches ( $L1, L2, L3, \dots$ )
    - TLB (cache for memory mapping)
  - **Result: Thread Context Switch is much Faster than Process Context Switch**
- 
- Saves time

# Putting it together: Process

## (Unix) Process



# Threads Context Switch



- Thread Switch overhead: low
  - Only CPU state switched
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low
  - No context switches
  - Only low-overhead thread-switches

# Threads in Use

- **Operating systems** need to be able to handle multiple things at once (MTAO)
  - processes, *interrupts*, background system maintenance
- **Servers** need to handle MTAO
  - *Multiple connections handled simultaneously in a Web Server*
- **Parallel programs** need to handle MTAO
  - To achieve better performance
- **Programs with user interfaces** often need to handle MTAO
  - To achieve user responsiveness while doing computation
- **Network and disk bound programs** need to handle MTAO
  - To hide network/disk latency

# There's a Problem!!!

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A

$x = 1;$

Thread B

$y = 2;$

- However, What about (Initially,  $y = 12$ ):

Thread A

$x = 1;$

$x = y+1;$

Thread B

$y = 2;$

$y = y^2;$

- *What are the possible values of  $x$ ?*

Thread A

$x = 1;$

$x = y+1;$

Thread B

$y = 2;$

$y = y^2$

x=13

# Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A

$x = 1;$

Thread B

$y = 2;$

- However, What about (Initially,  $y = 12$ ):

Thread A

$x = 1;$

$x = y+1;$

Thread B

$y = 2;$

$y = y^*2;$

- *What are the possible values of  $x$ ?*

Thread A

$x = 1;$   
 $x = y+1;$

Thread B

$y = 2;$   
 $y = y^*2;$

x=5

# Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn't matter:

Thread A

$x = 1;$

Thread B

$y = 2;$

- However, What about (Initially,  $y = 12$ ):

Thread A

$x = 1;$

$x = y+1;$

Thread B

$y = 2;$

$y = y^*2;$

- *What are the possible values of  $x$ ?*

Thread A

$x = 1;$   
 $x = y+1;$

Thread B

$y = 2;$

$y = y^*2;$

$x=3$

# Race Condition is the name!!

- **Race condition:** Output of a concurrent program depends on the **order of operations** between threads
- Cannot make any assumptions about relative speed of threads (i.e. interleaving is a given)
- **Non-determinism** is omnipresent:
  - *Scheduler's decision depends on many factors*
  - *Processor architecture (e.g., variable clock rate, out-of-order execution)*

# Race Condition for Instruction Set

- Simple threaded code (assume x=0)

Thread1

$x=x+1;$

Thread2

$x=x+1;$

Compiler Generated (because it can only use things from the Instruction Set):

*load r, x*  
*add r, r, 1*  
*store x, r*

Pre-emption

values of x can be 1 or 2 depending on the order of execution

*load r1, x*  
*add r1, r1, 1*  
*store x, r1*

*load r2, x*  
*add r2, r2, 1*  
*store x, r2*

X=2

*load r1, x*

*add r1, r1, 1*  
*store x, r1*

*load r2, x*  
*add r2, r2, 1*  
*store x, r2*

X=1

# Race Condition for Out-of-Order Execution

- Most architectures support (I should say “require”) this feature
  - *Pipelined processors cannot achieve peek performance without it*
- The idea is simple: compilers may reorder “unrelated” instructions like the following:

```
//global variables
bool done = false;
int data = -1;

Thread1 (){
    // takes a long time
    data = longfunction();
    done = true;
}
```

reorder

```
//global variables
bool done = false;
int data = -1;

Thread1 (){
    done = true;
    // takes a long time
    data = longfunction();
}
```

# Out-of-Order Execution causing Race Condition

- The problem arises when there is an inter-thread dependency
- Because of thread 2's wait, the 2 lines in Thread1() are no longer independent
- However, the compiler has no way to tell that!!!

```
// thread 1's function,  
// REORDERD  
Thread1 (){  
    done = true;  
    data = longfunction();  
}
```

```
// thread 2's function  
Thread2 (){  
    while (!done); // wait  
    int newdata = compute (data);  
}
```

# Thread API

- We will discuss thread API from C++11 which is cross-platform
  - *But for many other things, we will still use Linux for our PAs*
- Here is an example:

```
#include <iostream>
#include <thread>
#include <unistd.h>
using namespace std;

void foo() {
    sleep(3);
    cout<<"foo done"=><endl;
}
void bar(int x){
    sleep(1);
    cout <<"bar done"=><endl;
}
```

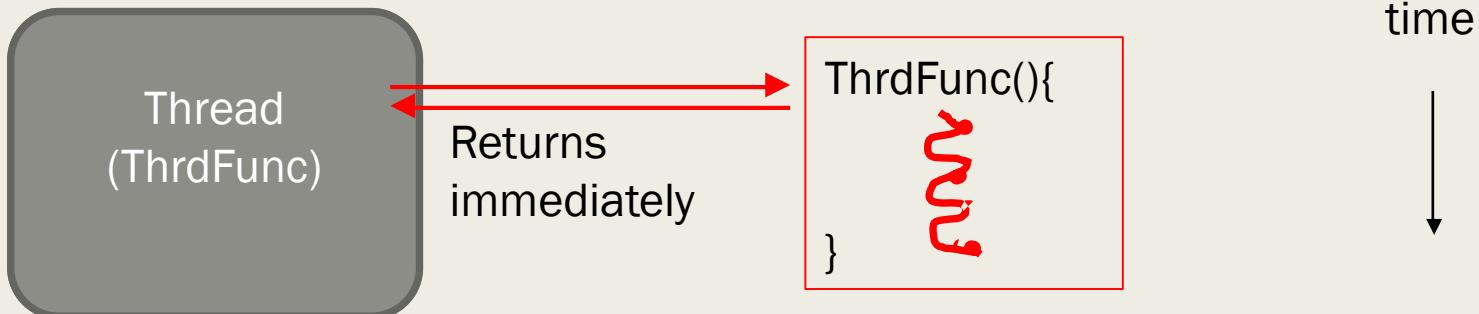
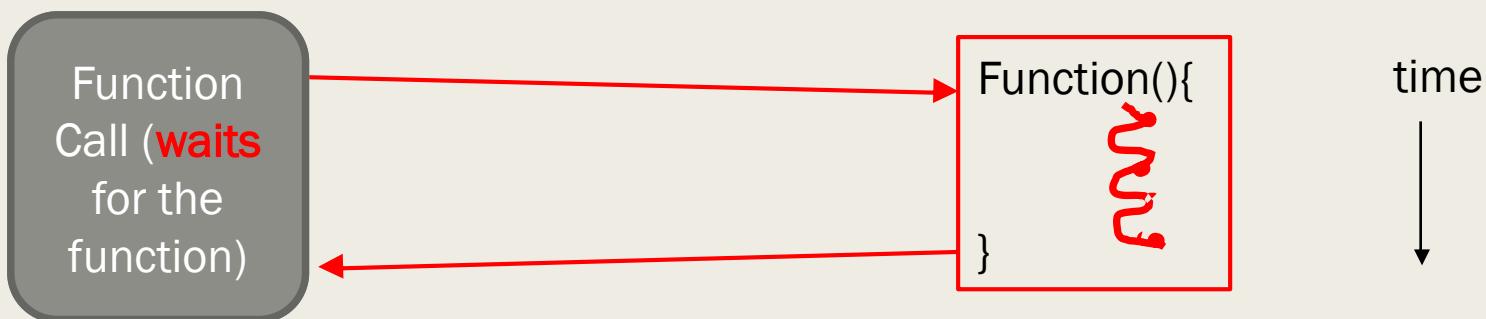
```
int main() {
    thread foothrd (foo); //calls foo() in a thread
    thread barthrd (bar,0); //calls bar(x) in a thread

    cout << "main, foo and bar would now execute
concurrently..." << endl;

    // synchronize threads:
    foothrd.join(); // pauses until foo finishes
    barthrd.join(); // pauses until bar finishes
    cout << "foo and bar completed.\n";
    return 0;
}
```

# Thread Execution

- Creating a thread is very similar to calling a function directly, with a slight difference shown below:
  - A regular function call *is blocking*, i.e., caller waits for callee
  - thread does **NOT**
- So, this is like fork(), creates the thread, calls the function inside, but does not wait



# Race Condition Demonstration

- Start 2 (or more) threads that increment a shared variable times
  - Use pointer to data so that threads share
- Use a large number for x to ensure adequate overlap
- Notice that the output is different every time

```
void func (int * p, int x) {
    // increment *p x times
    for (int i=0; i<x; i++)
        *p = *p + 1;
}
int main(int ac, char** av) {
    int data = 0;
    int times = atoi (av [1]);
    // start 2 thread to increment the same variable
    thread t1 (func, &data, times);
    thread t2 (func, &data, times);

    t1.join(); // pauses until first finishes
    t2.join(); // pauses until second finishes
    cout << "data = " << data << endl;
    return 0;
}
```

```
osboxes@osboxes:~/ $ ./a.out 10000000
data = 15003189
osboxes@osboxes:~/ $ ./a.out 10000000
data = 13380972
osboxes@osboxes:~/ $ ./a.out 10000000
data = 12565682
```

Reading Reference:

Textbook: Chapter 4

# THREAD SYNCHRONIZATION

Tanzir Ahmed  
CSCE 313 Spring 2020

# Goals for This Lecture

- Concurrency examples and sharing
- Synchronization
- Hardware Support for Synchronization

*Note: Some slides and/or pictures in the following are adapted and/or used verbatim from slide content in Silberschatz, Galvin, and Gagne (2014), Anthony D. Joseph (2014 Berkeley), Tom Anderson (2014 UW), Bettati (2014 TAMU), Gu (2014 TAMU), Tyagi (2016 TAMU)*

# Outline

- This lecture is a bit up-side down
- First, we learn how to **USE** Locks, Semaphores, Conditions
- Then, we will see how to **IMPLEMENT** Locks
  - *Programming Assignment 4 will ask you to implement Semaphores*
- The reason for this is to prepare you for PAs before exploring the nitty-gritty details

# Atomic Operations

- **Atomic Operation:** an operation that always runs to completion or not at all
  - *It is indivisible: it cannot be stopped in the middle and state cannot be modified by someone else in the middle*
  - *Fundamental building block – if no atomic operations, then have no way for threads to work together*
- Each instruction in the Instruction Set is atomic
  - *An instruction fully finishes before the current process/thread can be preempted/interrupted*

# Synchronization Variable – Lock to Provide Mutual Exclusion

- First step towards making shared data thread-safe
- The idea is to make instructions atomic to stop context switch from happening
- General Idea:

```
Lock();  
load r, x  
add r, r, 1  
store x, r  
Unlock();
```

Critical  
Section  
(No Context  
Switch)

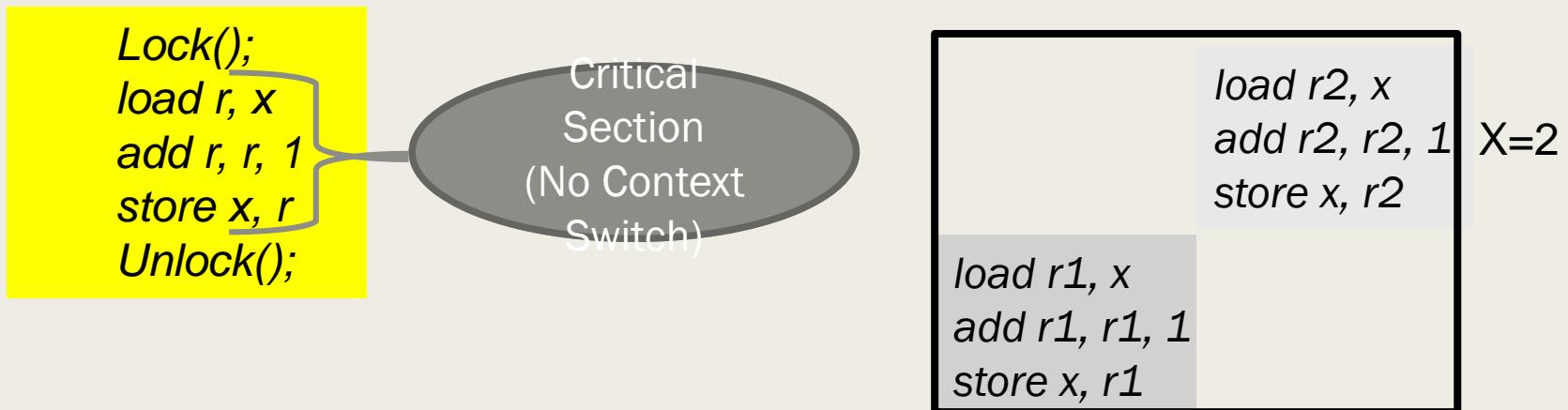
```
load r1, x  
add r1, r1, 1  
store x, r1
```

```
load r2, x  
add r2, r2, 1  
store x, r2
```

X=2

# Synchronization Variable – Lock to Provide Mutual Exclusion

- First step towards making shared variable thread-safe
- The idea is to make instructions atomic to stop context switch from happening
- General Idea:



# Mutex in C++ to Thread Safety

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

void func (int * p, int x, mutex* m) {
    // increment *p x times
    m->lock();
    for (int i=0; i<x; i++){
        *p = *p + 1;
    }
    m->unlock();
}

int main(int ac, char** av) {
    int data = 0;
    int times = atoi (av [1]);
    mutex m;

    thread t1 (func, &data, times, &m);
    thread t2 (func, &data, times, &m);

    t1.join(); // pauses until first finishes
    t2.join(); // pauses until second
               // finishes
    cout << "data = " << data << endl;
}
```

2. Lock it before the “critical section”

3. Unlock after “critical section”

1. Create a mutex and share it across the threads

```
osboxes@osboxes:~/\$ ./a.out 10000000
data = 20000000
osboxes@osboxes:~/\$ ./a.out 10000000
data = 20000000
osboxes@osboxes:~/\$ ./a.out 10000000
data = 20000000
```

# Mutex in C++ - Finer Locking

```
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

void func (int * p, int x, mutex* m) {
    // increment *p x times
    for (int i=0; i<x; i++){
        m->lock();
        *p = *p + 1;
        m->unlock();
    }
}

int main(int ac, char** av) {
    int data = 0;
    int times = atoi (av [1]);
    mutex m;

    thread t1 (func, &data, times, &m);
    thread t2 (func, &data, times, &m);

    t1.join(); // pauses until first finishes
    t2.join(); // pauses until second
               // finishes
    cout << "data = " << data << endl;
```

Critical  
Section

This is more  
fine-grained,  
produces the  
same correct  
result

- The previous approach puts entire thread under lock
- This effectively makes the threads completely sequential
  - *No threading/interleaving happens at all*
- This is “coarse-grained” locking
- You can make locking “finer” (see the example on left)
- The result is correct in both cases
- The choice would depend on other factors
  - *Locking and unlocking usually take time*

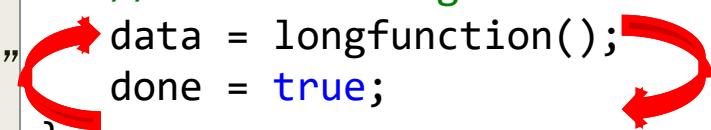
```
osboxes@osboxes:~/\$ ./a.out 10000000
data = 20000000
osboxes@osboxes:~/\$ ./a.out 10000000
data = 20000000
osboxes@osboxes:~/\$ ./a.out 10000000
data = 20000000
```

# Producer-Consumer Synchronization

- Just thread-safety is not adequate for many problems
- For instance, there is no order between operations in Thread 1 & 2 in the previous example:
- What if you want to Thread 1 to finish completely before you run Thread 2?
  - *Probably, Thread 1 produces a result that Thread 2 uses*
- This is called Producer-Consumer problem
- You need another synchronization primitive called “conditional variable”
  - *Because the naïve approach does not work for “instruction reordering”*

```
bool done = false;
int data = -1;
Thread1 (){
    // takes a long time
    data = longfunction();
    done = true;
}

// thread 2's function
Thread2 (){
    while (!done); // wait
    int newdata = compute (data);
}
```



# Producer-Consumer

```
bool done = false;  
int data = -1;  
condition_variable cv;  
mutex m;  
void Thread1 (){  
    data = longfunction();  
    m.lock();  
    done = true;  
    m.unlock();  
    cv.notify_all();  
}
```

```
void Thread2 (){  
    unique_lock<mutex> l (m);  
    cv.wait (l, []{return done == true;});  
  
    data = compute (data);  
    cout << "Data is: " << data << endl;  
  
    l.unlock();  
}
```

- Step 1: Declare a condition variable and a mutex
- Step 2: The producer
  - *produces data*
  - Set the predicate variable (*done = true*) under lock
  - Calls *notify\_one/all()* on the condition to wake up the consumers

# Producer-Consumer

```
bool done = false;  
int data = -1;  
condition_variable cv;  
mutex m;  
void Thread1 (){  
    data = longfunction();  
    m.lock();  
    done = true;  
    m.unlock();  
    cv.notify_all();  
}
```

```
void Thread2 (){  
    unique_lock<mutex> l (m);  
    cv.wait (l, []{return done == true;});  
  
    data2 = compute (data);  
    cout << "Data is: " << data << endl;  
  
    l.unlock();  
}
```

- Step 3: The consumer(s) do the following:
  - *Calls wait() on the condition*
  - *Wait() needs a “wrapped” lock (as unique\_lock) and a predicate function*
    - The “wrapper” also locks the lock
    - In the above I just used a “lambda” function for the predicate instead of defining it elsewhere
  - *Consume data*
  - *Unlock the lock*

# Another Example of Producer-Consumer

```
bool done = false;
Queue<int> q;
condition_variable cv;
mutex m;
void Producer (x){
    m.lock();
    q.push (x);
    m.unlock();
    cv.notify_one();
}
```

```
void Consumer (){
    unique_lock<mutex> l (m);
    cv.wait (l, []{return q.size() > 0;});

    int data = q.pop();
    cout << "Data is: " << data << endl;

    l.unlock();
}
```

# A Few More Points on Wait()

- First, the cv.wait() function wakes up the waiting threads sequentially, NOT all at once
  - *This provides a Critical Section until unlock() function is called on the lock later on*
  - *As a result, every awake thread gets to do something without running into race condition*
- The wait() function internally unlocks the lock before going to sleep()
  - *Otherwise, even the producer cannot produce, which requires the lock itself*
  - *However, when returning from wait(), the lock is grabbed again and that causes the “one-at-a-time” safety*
- Spurious wake ups are possible
  - *Waiting threads may wake up even when the predicate is not true*
  - *That's why using the “predicate” is absolute necessity*

# Producer-consumer with a bounded buffer

- Problem Definition

- Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer

- Don't want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them

- Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

- Example: Coke machine

- Producer can put limited number of cokes in machine
  - Consumer can't take cokes out if machine is empty



# Correctness constraints for solution

- Correctness Constraints:
  - *Consumer must wait for producer to fill slots, if empty (scheduling constraint)*
  - *Producer must wait for consumer to make room in buffer, if all full (scheduling constraint)*
  - *Only one thread can manipulate buffer queue at a time (mutual exclusion using lock)*
- Nice Features
  - *Inherent rate control:*
    - Consumer is limited by Production Rate
    - Producer is limited by buffer size and consequently Consumption Rate
- Application is universal
  - *Networks, Inter Process Communication etc.*

# Implementing BoundedBuffer

The following is a skeleton for the BoundedBuffer, that is:

Unsafe: because multiple threads can push into and pop from it simultaneously, leading to race condition

Unbounded: because nothing stops from the buffer from growing to infinity when producer thread(s) is(are) much faster than the consumer(s)

We need to implement thread-safety and bounds on overflow and underflow

- We also want efficient wait until the overflow/underflow conditions are gone, so Producer/Consumer do not have to “retry”

```
/* a unsafe and unbounded buffer */
/* add necessary changes */
class BoundedBuffer{
    queue<vector<char>> q;
    int maxcap; // max capacity
    // add sync. variables here
public:
    BoundedBuffer (int _cap) :maxcap(_cap){
        vector<char> pop (){
            vector<char> data = q.front();
            q.pop();
            return data;
        }
        void push (vector<char> data){
            q.push (data);
        };
    }
}
```

# Let's Implement BounderBuffer

- Note that we need to implement a sophisticated wait facility
- The user will call pop() or push() only once, your function will wait until the time is right for that operation
  - *If the buffer is full, make the push() function wait until there is space (because somebody popped something out)*
  - *If the buffer is empty, pop() function waits until somebody pushes something*
    - Of course, if many pop() functions are waiting and only 1 push happens, only 1 pop() function will get out of sleep
    - Similar reasoning for push functions as well
- Here is a naïve implementation that will NOT work:

```
vector<char> BoundedBuffer::pop () {
    while (buffer.size () == 0)
        sleep (n);
    // now consume
}
```

There is a race condition

# BounderBuffer – Take 2

```
vector<char> BoundedBuffer::pop () {
    mtx.Lock () ; // mtx is a mutex, defined
                  as class member variable
    while (buffer.size () == 0)
        sleep (n) ;
    mtx.Unlock () ;
    // now consume
}
```

No unwanted switches now. But, sleeping with  
Mutex Locked!!! Even the producers cannot  
replenish buffer while this thread is waiting

# Take 3

```
Vector<char> BoundedBuffer::pop () {
    mtx.Lock ();
    while (buffer.size () == 0) {
        mtx.Unlock ();
        // 1. thread switch???
        sleep (n);
        // 2. thread switch???
        mtx.Lock ();
    }
    mtx.Unlock ();
}
```

Producer  
thread  
runs, you  
just missed  
it

Another  
consumer  
takes it,  
again missed

# Condition Variables for BB

- Problems with the previous solution:
  - *Misses wakeups*
  - *Does busy-looping (takes CPU cycles away from Producers)*
  - *Sleep(x) is dependent on x. What if x=1sec, and data is produced every 5ms? The consumer is not very responsive!!*
  - *What if x=1ms and data are produced every 10 sec. The consumer is taking up unnecessary CPU cycles, possibly away from the Producer*
  - *In summary, a synchronous solution is not elegant*
- This is exactly where condition variables are absolutely essential

# BoundedBuffer using Condition Variables

```
class BoundedBuffer
{
private:
    int cap;
    queue<vector<char>> q; /*queue where each item is a vector
of char so that we can store variable length data */

    /*mutex to protect the queue from simultaneous producer
accesses or simultaneous consumer accesses */
    mutex mtx;

/*cond. that tells the consumers that some data is there */
    condition_variable data_avail;
/*cond. that tells the producers some slot is available */
    condition_variable slot_avail;
```

# Condition Variables for BB – A working (almost) solution

```
vector<char> BoundedBuffer::pop(){
    unique_lock<mutex> l (mtx);
    // keep waiting as long as q.size() == 0
    data_avail.wait (l, [this]{return q.size() > 0;});
    // pop from the queue
    vector<char> data = q.front();
    q.pop();
    // notify any potential producer(s)
    slot_avail.notify_one ();
    // unlock the mutex so that others can go in
    l.unlock();
    return data;
}
```

1. Atomically unlocks the mutex and goes to sleep
2. Returns when condition set and with mutex locked
3. Return does not mean buffer has something
4. Condition is set by the Producer
5. Only 1 consumer will consume, others will wait for the next Production

# Summary So Far

- We have learned about 2 very important synchronization primitives:
  - *Mutexes*
  - *Condition variables*
- We should be able to solve most synchronization problems using these two
  - *Let us try a few synchronization problems to convince ourselves*
- There are other more convenient primitives that can be built using Mutex and Condition variable. They can be very convenient and developer friendly
  - *Example: Semaphores, Synchronized methods in Java*

# Semaphores



- Semaphores are a kind of generalized locks
  - *First defined by Dijkstra in late 60s*
  - *Main synchronization primitive used in original UNIX*
- The main purpose is to make our code simpler from what we saw before using condition variables and mutex
  - *We need to use a mutex and a condition variable just to indicate one event*
  - *This makes our code complicated*
- Definition: a Semaphore has a **non-negative integer value** and supports the following two **atomic** operations:
  - P(): Waits until value > 1, then decrements it by 1
    - Think of this as the wait() operation, or the lock() operation
  - V(): Increments value by 1
    - May wake up a waiting P, if any
    - Think of this as the signal() operation, or the unlock() operation

# Semaphore Implementation

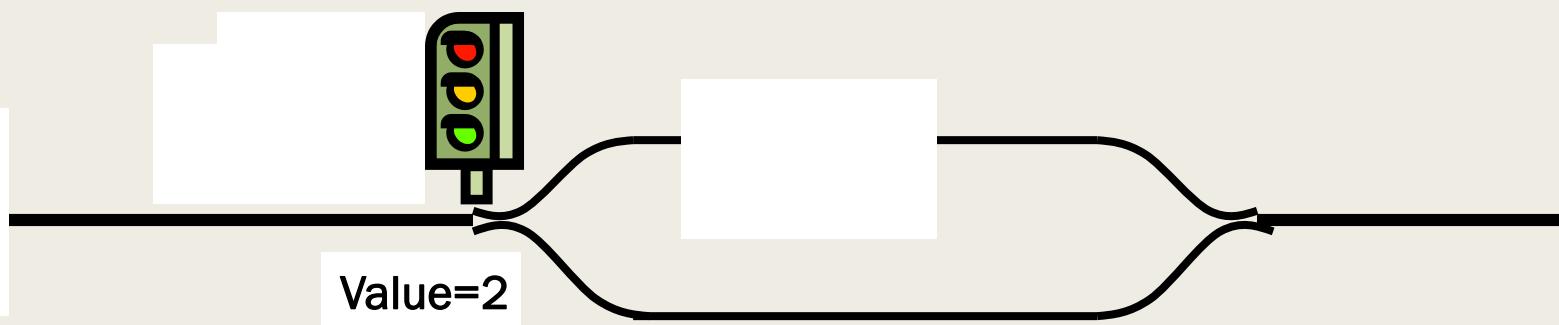
```
class Semaphore{  
private:  
    int value;  
    mutex m;  
    condition_variable cv;  
public:  
    Semaphore (int _v):value(_v){}  
    void P(){  
        unique_lock<mutex> l (m);  
        // wait until the value is positive  
        cv.wait (l, [this]{return value > 0;});  
        // now decrement  
        value --;  
        l.unlock (); // this is optional  
    }  
    void V(){  
        unique_lock<mutex> l (m);  
        value ++;  
        cv.notify_one();  
        l.unlock(); // this is optional  
    }  
};
```

Wait until  $\text{value} > 0$ , so it can be decremented. Then decrement it.

Always notify on the way out, but do not `notify_all()` to avoid spurious wake ups

# Semaphores Like Integers Except

- No negative values
- Only operations allowed are P and V – can't read or write value, except to set it initially
- Operations must be atomic
  - *Two P's together can't decrement value below zero*
  - *Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time*
- Semaphore from railway analogy
  - *Here is a semaphore initialized to 2 for resource control:*



# Two Uses of Semaphores

## ■ Mutual Exclusion (initial value = 1)

- Also called “Binary Semaphore”.
- Can be used for mutual exclusion:

```
semaphore.P();  
// Critical section goes here  
semaphore.V();
```

## ■ Scheduling Constraints (initial value = 0)

- Allow *thread 1* to wait for a signal from *thread 2*, i.e., *thread 2 schedules thread 1* when a given **constrained** is satisfied
- Example: suppose you had to implement *ThreadJoin* which must wait for a thread to terminate:

```
Initial value of semaphore = 0  
  
ThreadJoin {  
    semaphore.P();  
}  
  
ThreadFinish {  
    semaphore.V();  
}
```

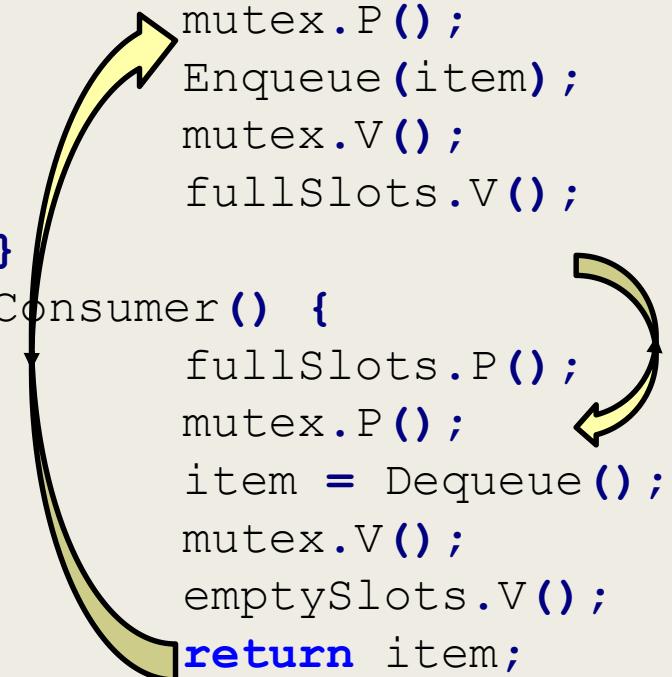


# Full Solution to Bounded Buffer

```
Semaphore fullSlots = 0;    // Initially, no coke
Semaphore emptySlots = bufSize; // Initially all empty
Semaphore mutex = 1;          // No one using machine

Producer(item) {
    emptySlots.P();           // Wait until space
    mutex.P();                // Wait until machine free
    Enqueue(item);
    mutex.V();
    fullSlots.V();            // Notify there is more coke
}

Consumer() {
    fullSlots.P();            // Check if there's a coke
    mutex.P();                // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptySlots.V();           // tell producer need more
    return item;
}
```



# Thoughts

## ■ Why asymmetry?

- *Producer does:*  $emptySlots.P()$ ,  
 $fullSlots.V()$

Decrease # of  
empty slots

Increase # of  
occupied slots

- *Consumer does:*  $fullSlots.P()$ ,  
 $emptySlots.V()$

Decrease # of  
occupied slots

Increase # of  
empty slots

One is creating space, the other is filling space

# More Thoughts

- Is order of P's important?
- Is order of V's important?
- What if we have 2 producers or 2 consumers?

```
Producer(item) {  
    emptySlots.P();  
    mutex.P();  
    Enqueue(item);  
    mutex.V();  
    fullSlots.V();  
}  
  
Consumer() {  
    fullSlots.P();  
    mutex.P();  
    item = Dequeue();  
    mutex.V();  
    emptySlots.V();  
    return item;  
}
```

# More Thoughts

- Is order of P's important?
  - Yes! Can cause deadlock

BEFORE

```
Producer(item) {  
    emptySlots.P();  
    mutex.P();  
    Enqueue(item);  
    mutex.V();  
    fullSlots.V();  
}  
  
Consumer() {  
    fullSlots.P();  
    mutex.P();  
    item = Dequeue();  
    mutex.V();  
    emptySlots.V();  
    return item;  
}
```

AFTER

```
Producer(item) {  
    mutex.P();  
    emptySlots.P();  
    Enqueue(item);  
    mutex.V();  
    fullSlots.V();  
}  
  
Consumer() {  
    fullSlots.P();  
    mutex.P();  
    item = Dequeue();  
    mutex.V();  
    emptySlots.V();  
    return item;  
}
```

# More Thoughts

- Is order of V's important?
  - No, except that it might affect scheduling efficiency

BEFORE

```
Producer(item) {  
    emptySlots.P();  
    mutex.P();  
    Enqueue(item);  
    mutex.V();  
    fullSlots.V();  
}  
  
Consumer() {  
    fullSlots.P();  
    mutex.P();  
    item = Dequeue();  
    mutex.V();  
    emptySlots.V();  
    return item;  
}
```

AFTER

```
Producer(item) {  
    emptySlots.P();  
    mutex.P();  
    Enqueue(item);  
    fullSlots.V();  
    mutex.V();  
}  
  
Consumer() {  
    fullSlots.P();  
    mutex.P();  
    item = Dequeue();  
    mutex.V();  
    emptySlots.V();  
    return item;  
}
```

# More Thoughts

- What if we have 2 producers or 2 consumers?
  - *Do we need to change anything?*
    - NO

```
Producer(item) {  
    emptySlots.P();  
    mutex.P();  
    Enqueue(item);  
    mutex.V();  
    fullSlots.V();  
}  
  
Consumer() {  
    fullSlots.P();  
    mutex.P();  
    item = Dequeue();  
    mutex.V();  
    emptySlots.V();  
    return item;  
}
```

# Some Classic Synchronization Problems

- Reader-Writer problem
- Dining-Philosopher Problem
- Monkey crossing bridge

# Readers-Writers Problem

- Many processes share a database
- Some processes write to the database
- Only one writer can be active at a time
- Any number of readers can be active simultaneously
- First Readers-Writers Problem:
  - *Readers get higher priority, and do not wait for a writer*
- Second Readers-Writers Problem:
  - *Writers get higher priority over Readers waiting to read*
    - Courtois et al.

# Readers-Writers

```
Semaphore wrl (1); // to
exclude readers and writers
int rcount = 0; // to count
how many active readers
Semaphore mutex(1); // to
protect rcount variable

void Writer () {
    do {
        wrl.P();
        /*writing is performed*/
        wrl.V();
    }while(true);
}
```

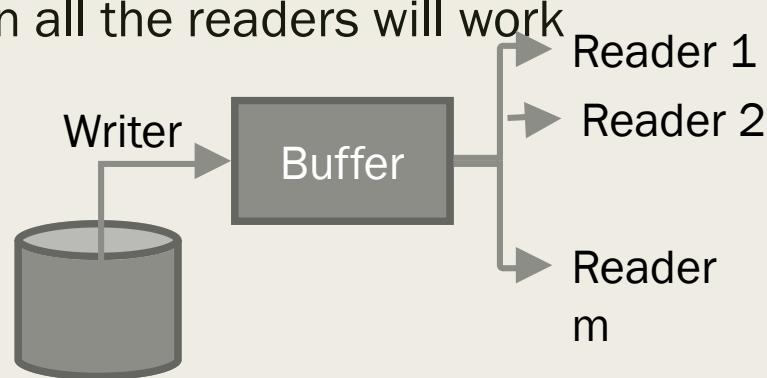
```
void Reader () {
    do {
        //maintain rcount, but safely
        mutex.P();
        rcount++;
        if (rcount == 1)
            wrl.P(); // wait for writer
        mutex.V();
        //now going in with the lock
        at hand, no writer here
        /*reading is performed*/
        // maintain rcount, but
        safely
        mutex.P();
        rcount--;
        if (rcount == 0)
            wrl.V(); //last reader, so
        give up lock
        mutex.V();
    }while(true);
}
```

# Readers-Writers Notes

- If there is a writer
  - *First reader blocks on **wrl***
  - *Other readers block on **mutex***
- Once a writer exists, all readers get to go through
  - *Which reader gets in first?*
- The last reader to exit signals a writer
  - *If no writer, then readers can continue*
- If readers and writers waiting on **wrl**, and writer exits
  - *Who gets to go in first?*
- Why doesn't a writer need to use **mutex**?

# How about a more Specific Reader-Writer problem?

- Problem: One writer thread in charge of reading from an external source (e.g., disk or network) and putting data in a buffer
- $m$  reader threads who consume from the buffer
- First, the writer will populate data, then all the readers will work
- Does this problem look familiar?
- BoundedBuffer with buffer size 1



- Let us see some similar examples in action

# Monkeys-Crossing-River Problem

- Several monkeys trying to cross a river on a thin rope
  - *The rope can carry at most  $M$  monkeys at a time*
- You have to fill-up the following functions used by each monkey:

```
void monkey(int monkeyid) {  
    WaitUntilSafe();          // define  
    CrossRiver(monkeyid); // given  
    DoneWithCrossing();     // define  
}
```

- Will 1 semaphore suffice?

# Follow-Up

- What if now complicate the problem a little bit
- New Restrictions:
  - *There are 2 directions the monkeys are moving (east and west)*
  - *All monkeys executing in CrossRiver() are heading in the same direction.*
- Now, the function looks like the following:

```
void monkey(int monkeyid, int dir) {  
    WaitUntilSafe(dir);           // define  
    CrossRavine(monkeyid, dir); // given  
    DoneWithCrossing(dir);      // define  
}
```

# Monkeys Crossing with Directions

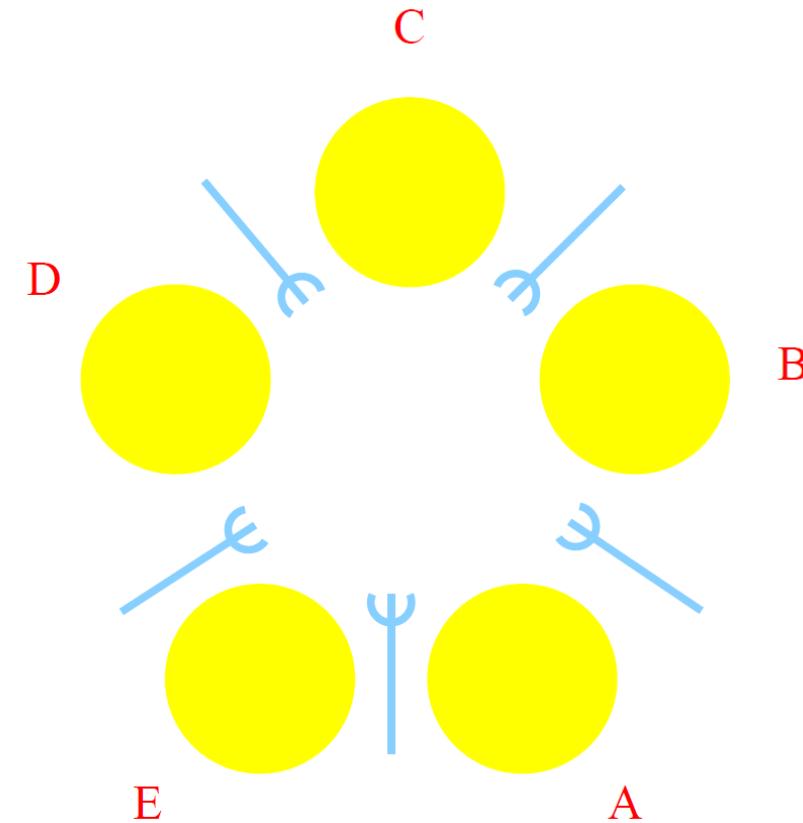
```
int monkey_count[2] = {0,0}; //counter for each direction
Semaphore mutex[2] = {1,1}; //mutexes to protect counters
Semaphore capacity(M); //ensure maximum M monkeys on rope
Semaphore dirmtx(1); // direction mutex used by the first
monkey from each direction
```

```
void WaitUntilSafe (int dir)
{
    mutex[dir].P();
    monkey_count[dir]++;
    if (monkey_count[dir]==1)
        dirmtx.P();
    mutex[dir].V();
    capacity.P();
}
```

```
void DoneWithCrossing(int dir)
{
    mutex[dir].P();
    monkey_count[dir]--;
    if (monkey_count[dir] == 0)
        //last mon:release dirmtx
        dirmtx.V();
    mutex[dir].V();
    capacity.V();
}
```

# Dining-Philosopher Problem

- There are a number of philosophers, who either 1) think or 2) eat spaghetti
- Each philosopher needs 2 forks to eat
- Have to avoid deadlock or starvation
  - *Every philosopher grabbed 1 fork to the left (or right)*
- This is left for your own study

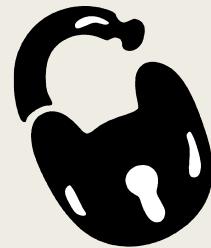


# Definitions and Quick Recap

- **Synchronization:** using atomic operations to ensure cooperation between threads
  - *For now, only loads and stores are atomic*
- **Critical Section:** piece of code that only one thread can execute at once
- **Mutual Exclusion:** ensuring that only one thread executes critical section
  - *One thread excludes the other while doing its task*
  - *Critical section and mutual exclusion are two ways of describing the same thing*

# More Definitions

- **Lock:** prevents someone from doing something
  - *Lock before entering critical section and before accessing shared data*
  - *Unlock when leaving, after accessing shared data*
  - *Wait if locked*
    - Important idea: all synchronization involves waiting



# Implementing a Lock

- How can we build multi-instruction atomic operations?
  - *Recall: dispatcher gets control in two ways.*
    - Internal: Thread does something to relinquish the CPU
    - External: Interrupts cause dispatcher to take CPU
  - *On a uniprocessor, can avoid context-switching by:*
    - Avoiding internal events
    - Preventing external events by disabling interrupts
- Consequently, naïve Implementation of locks:

```
LockAcquire { disable Ints; }
```

```
LockRelease { enable Ints; }
```

# Naïve use of Interrupt Enable/Disable: Problems

- Can't let user do this! Consider following:

```
LockAcquire() ;  
while(TRUE) { ; }
```

- Real-Time system—no guarantees on timing!
  - *Critical Sections might be arbitrarily long*
- What happens with I/O or other important events?
  - *“Reactor about to meltdown. Help?”*



# Better Implementation of Locks

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

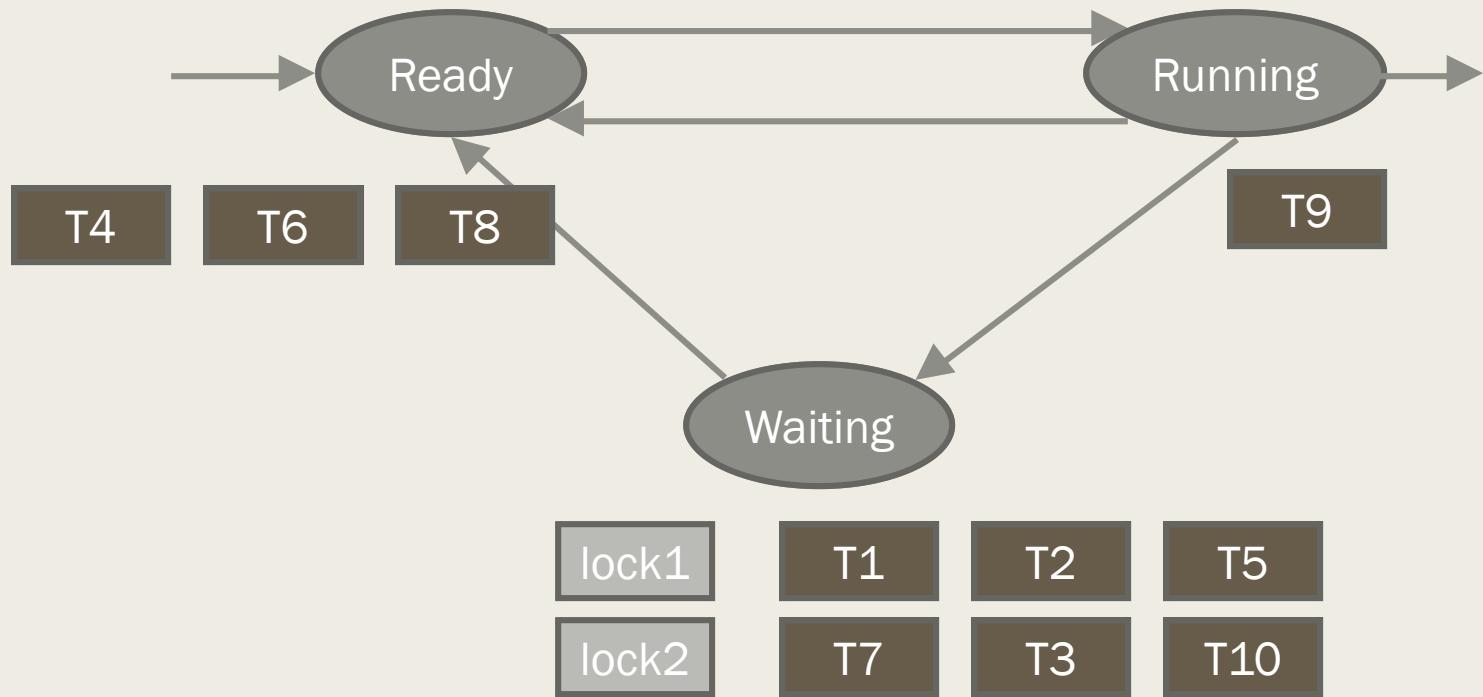
```
int value = FREE;

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
    } else {
        value = BUSY;
    }
    enable interrupts;
}
```

```
Release() {
    disable interrupts;
    if (anyone on wait queue) {
        take thread off wait queue
        Put on the ready queue
    } else {
        value = FREE;
    }
    enable interrupts;
}
```



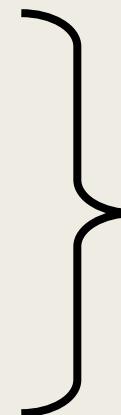
# Over-All Idea



# New Lock Discussion

- Disable interrupts: avoid interrupting between checking lock value (in Acquire()) and setting it (in Release())
  - *Otherwise two threads could think that they both have lock*

```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        Go to sleep();  
        // Enable interrupts?  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

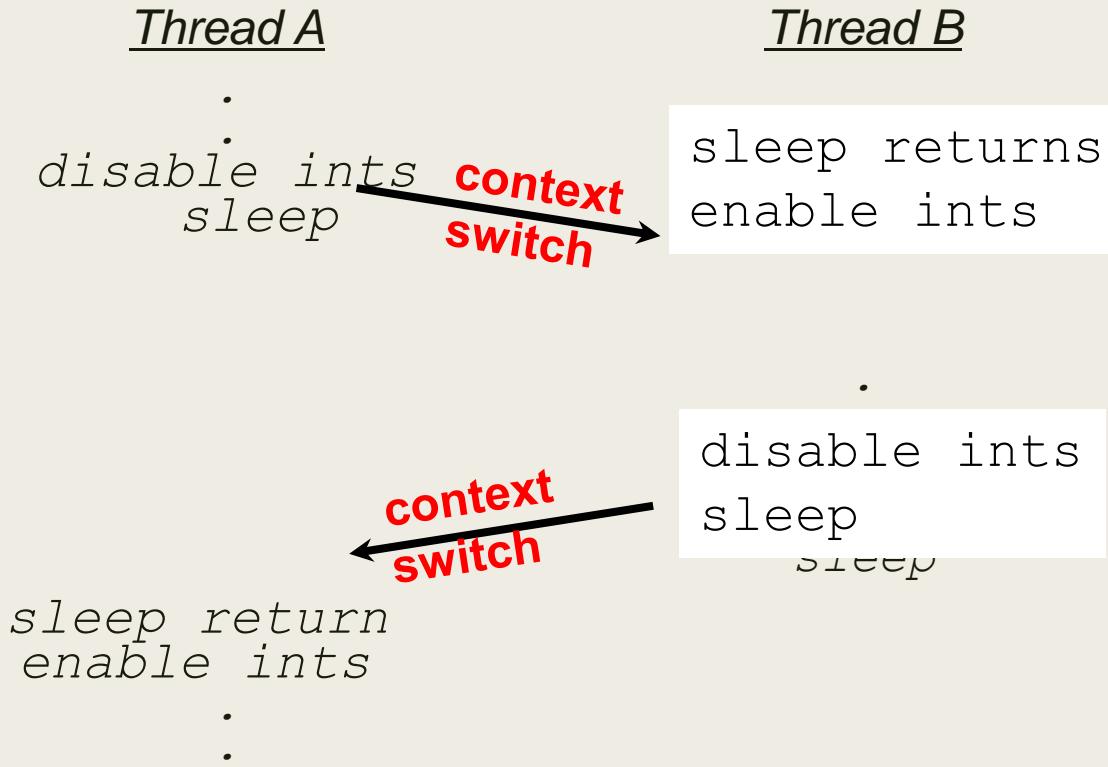


Critical  
Section

- Note: unlike previous solution, critical section very short
  - *User of lock can take as long as they like in their own critical section*
  - *Critical interrupts taken in time*

# Who Enables Interrupts then?

- Since ints are disabled when you call sleep:
  - *Responsibility of the next thread to re-enable ints*
- When the sleeping thread wakes up, returns to acquire and re-enables interrupts



# An Execution Trace

T1	T2	T3	Interrupt	Wait Queue for this lock	Ready Queue
Acquire()			Enabled		
	Acquire()		Disabled	T2	
CS of T1			Disabled	T2	
CS of T1			Disabled	T2	
Release()	Acquire() returns		Enabled		T2
		Acquire()	Disabled	T3	
	CS of T2		Disabled	T3	
	Release()		Enabled		T3

# Interrupt Re-enable in Going to Sleep

- What about re-enabling interrupts when going to sleep?

```
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        go to sleep();  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}
```

```
Release() {  
    disable interrupts;  
    if (anyone on wait queue) {  
        take thread off wait queue  
        Put on the ready queue  
    } else {  
        value = FREE;  
    }  
    enable interrupts;  
}
```

- Before putting thread on the wait queue?
  - *Thread switch to Release, which checks the queue does not find anybody and not wake up thread until next lock acquire/release*
- After putting the thread on the wait queue
  - *Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep*
  - *Misses wakeup and still holds lock (deadlock!) while sleeping*

# Summary So Far

- Locks showed so far require Interrupts enable/disable
  - *Separate lock variable reduces Interrupt Disabled duration*
  - *But that duration can still be intolerable*
- We need a better lock implementation
- Things are going towards the “usual suspect” – We are going to need **hardware support**

# Atomic Read-Modify-Write instructions

- Problems with interrupt-based lock solution:
  - *Can miss critical Interrupts while INT disabled*
  - *Can't give lock implementation to users*
  - *Doesn't work well on multiprocessor*
  - *Disabling interrupts on all processors requires messages and would be very time consuming*
- Alternative: atomic instructions added to the instruction set
  - *These instructions read a value from memory and write a new value atomically*
  - *Hardware is responsible for implementing this correctly*
    - on uniprocessors (not too hard)
    - and multiprocessors (requires help from cache coherence protocol)

# Examples of Read-Modify-Write

- ```
test&set (&address) /*most architectures*/
    result = M[address];
    M[address] = 1;
    return result;
}
```
- ```
swap (&address, register) { /* x86 */
    temp = M[address];
    M[address] = register;
    register = temp;
}
```

# Implementing Locks with test&set

```
int value = 0; // Free
Acquire() {
    while (test&set(value)); // while busy
}
Release() {
    value = 0;
}
```

```
test&set (&address) {
    result = M[address];
    M[address] = 1;
    return result;
}
```

Explanation:

- *If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits*
- *If lock is busy, test&set reads 1 and sets value=1 (no change). It returns 1, so while loop continues*
- *When we set value = 0, someone else can get lock*

# Problem: Busy-Waiting for Lock

- Positives for this solution
  - *Machine can receive interrupts*
  - *User code can use this lock*
  - *Works on a multiprocessor*
- Negatives
  - *Inefficient: busy-waiting thread will consume cycles*
  - *Waiting thread may take cycles away from thread holding lock!*
  - ***Priority Inversion:*** *If busy-waiting thread has higher priority than thread holding lock  $\Rightarrow$  no progress!*
- Priority Inversion problem with original Martian rover



# Better Locks using test&set

- Can we build test&set locks without busy-waiting?
  - *Can't entirely, but can minimize!*
  - *Idea: only busy-wait to atomically check lock value*

```
int guard = 0; //protects "value"
int value = FREE;
Acquire() {
    // Short busy-wait time
    while (test&set(guard));
    if (value == BUSY) {
        put thread on wait queue;
        go to sleep() & guard = 0;
    } else {
        value = BUSY;
        guard = 0;
    }
}
```

```
Release() {
    // Short busy-wait time
    while (test&set(guard));
    if (anyone on wait queue) {
        take thread off wait queue;
        Place on ready queue;
    } else {
        value = FREE;
    }
    guard = 0;
}
```

Note: sleep has to be sure to reset the guard variable

# Locks using test&set vs. Interrupts

- Compare to “disable interrupt” solution

```
int value = FREE;  
  
Acquire() {  
    disable interrupts;  
    if (value == BUSY) {  
        put thread on wait queue;  
        Go to sleep();  
        // Enable interrupts?  
    } else {  
        value = BUSY;  
    }  
    enable interrupts;  
}  
  
Release() {  
    disable interrupts;  
    if (anyone on wait queue) {  
        take thread off wait queue;  
        Place on ready queue;  
    } else {  
        value = FREE;  
    }  
    enable interrupts;  
}
```



- Basically replace
  - *disable interrupts* → `while(test&set(guard));`
  - *enable interrupts* → `guard = 0;`

# Recap: Locks

```
lock.Acquire();  
...  
critical section;  
...  
lock.Release();
```

```
Acquire() {  
    disable interrupts;  
}
```

```
Release() {  
    enable interrupts;  
}
```

If one thread in critical section, no other activity (including OS) can run!

```
int value = 0;  
Acquire() {  
    // Short busy-wait time  
    disable interrupts;  
    if (value == 1) {  
        put thread on wait-queue;  
        go to sleep()  
    } else {  
        value = 1;  
    }  
    enable interrupts;  
}  
  
Release() {  
    // Short busy-wait time  
    disable interrupts;  
    if anyone on wait queue {  
        take thread off wait-queue  
        Place on ready queue;  
    } else {  
        value = 0;  
    }  
    enable interrupts;  
}
```

# Recap: Locks

```
lock.Acquire();  
...  
critical section;  
...  
lock.Release();
```

```
int value = 0;  
Acquire() {  
    while(test&set(value));  
}
```

```
Release() {  
    value = 0;  
}
```

Threads waiting to  
enter critical section  
busy-wait

```
int guard = 0;  
int value = 0;  
Acquire() {  
    // Short busy-wait time  
    while(test&set(guard));  
    if (value == 1) {  
        put thread on wait-queue;  
        go to sleep() & guard = 0;  
    } else {  
        value = 1;  
        guard = 0;  
    }  
}  
  
Release() {  
    // Short busy-wait time  
    while (test&set(guard));  
    if anyone on wait queue {  
        take thread off wait-queue  
        Place on ready queue;  
    } else {  
        value = 0;  
    }  
    guard = 0;  
}
```

# INTER-PROCESS COMMUNICATION

Tanzir Ahmed  
CSCE 313 Spring 2020

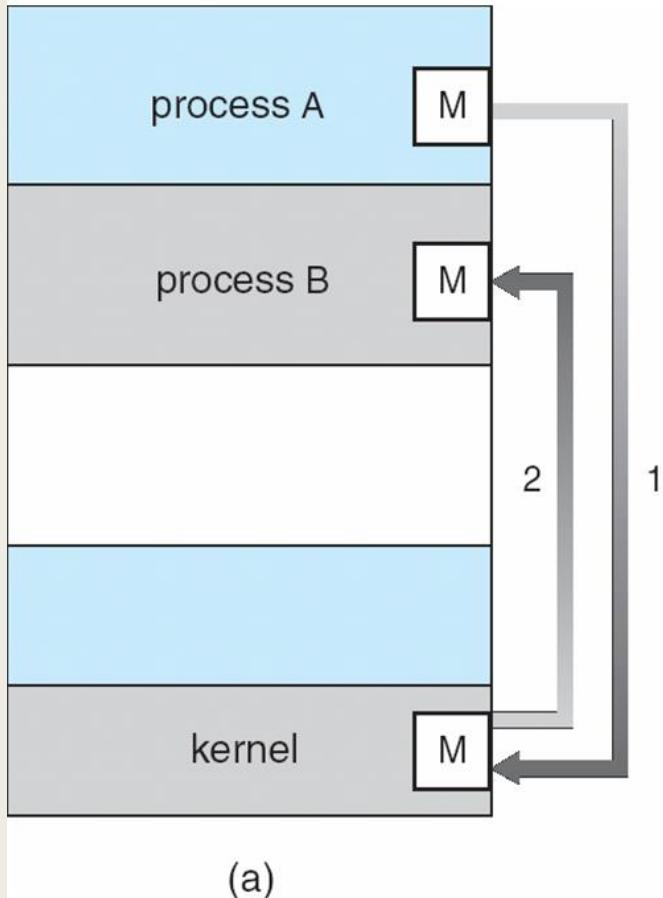
# Inter-Process Communication

- IPC Methods
  - *Pipes and FIFO*
  - *Message Passing*
  - *Shared Memory*
  - *Semaphore Sets*
  - *Signals*
- References:
  - Beej's guide to Inter Process Communication for the code examples (<https://beej.us/guide/bgipc/>)
  - *Understanding Unix/Linux Programming*, Bruce Molay, Chapters 10, 15
  - [Advanced Linux Programming Ch 5](#)
  - Some material also directly taken or adapted with changes from [Illinois course in System Programming](#) (Prof. Angrave), UCSD (Prof. Snoeren), and [USNA](#) (Prof. Brown)

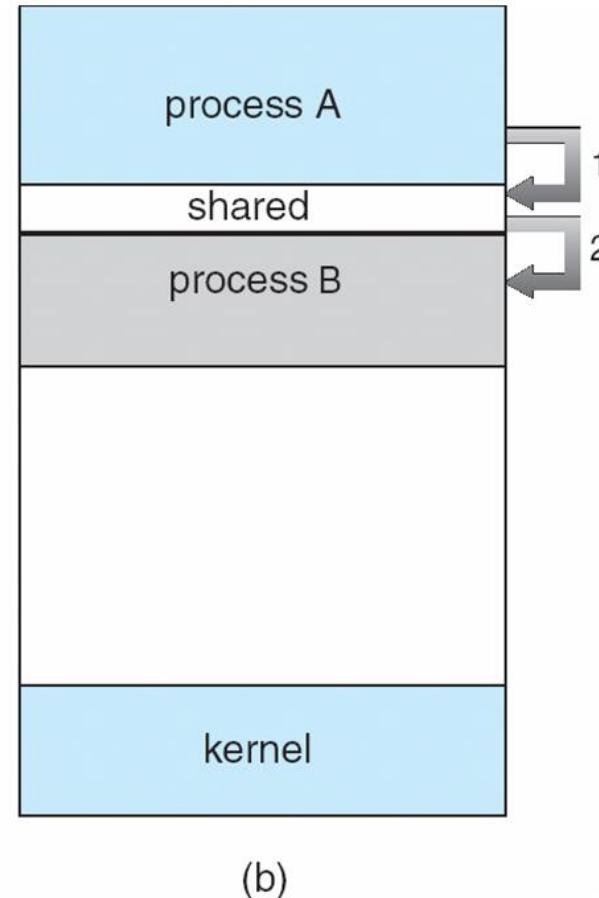
# Inter-Process Communication

- OS provides generic mechanisms to communicate
  - *Un-named and Named Pipes*
    - Explicit communication channel
    - Explicit Synchronization: read() operation is blocking
  - *Message Passing: explicit communication channel provided through send()/receive() system calls*
    - Explicit Synchronization through send() and recv()
  - *Shared Memory: multiple processes can read/write same physical portion of memory; implicit channel*
    - Implicit channel
    - No OS mediation required once memory is mapped
    - No synchronization between communicating threads
      - *There is no read/send or write/recv available*
      - *How to know when data is available?*

# IPC Fundamental Communication Models

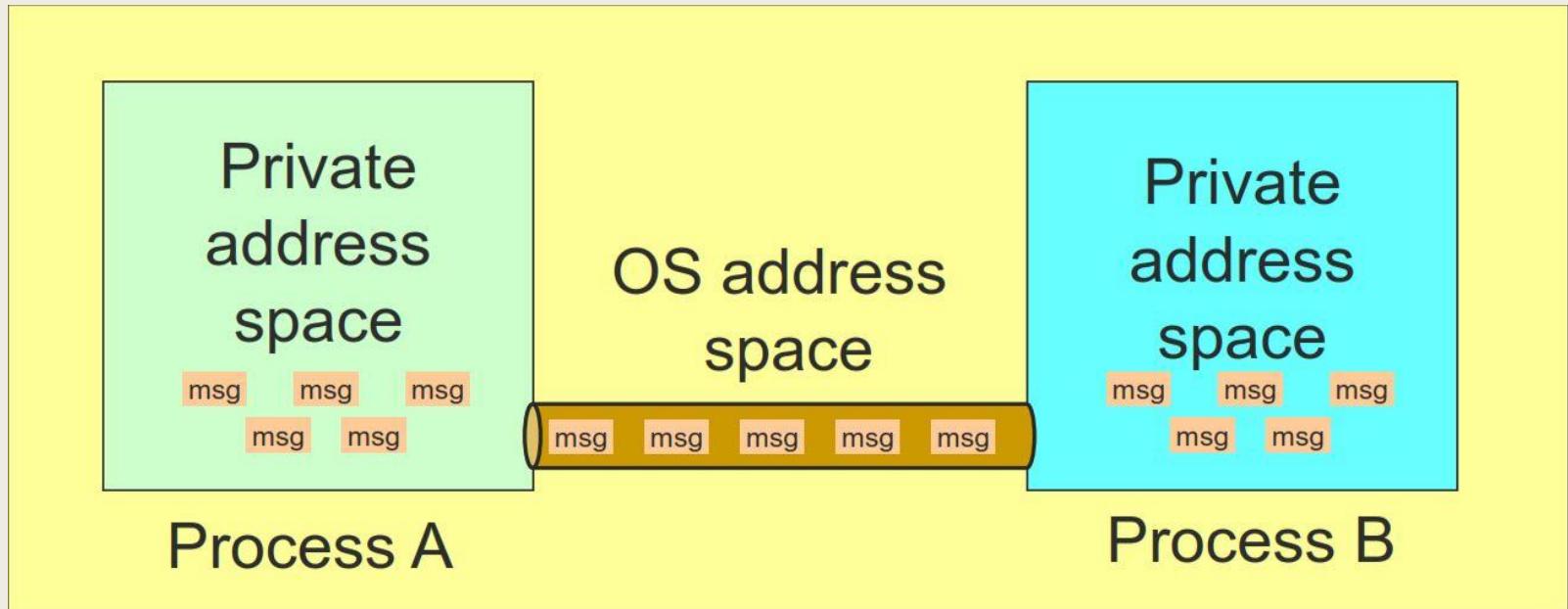


Example: pipe, fifo, message,  
signal



Example: shared memory, memory  
mapped file

# Communication Over a Pipe



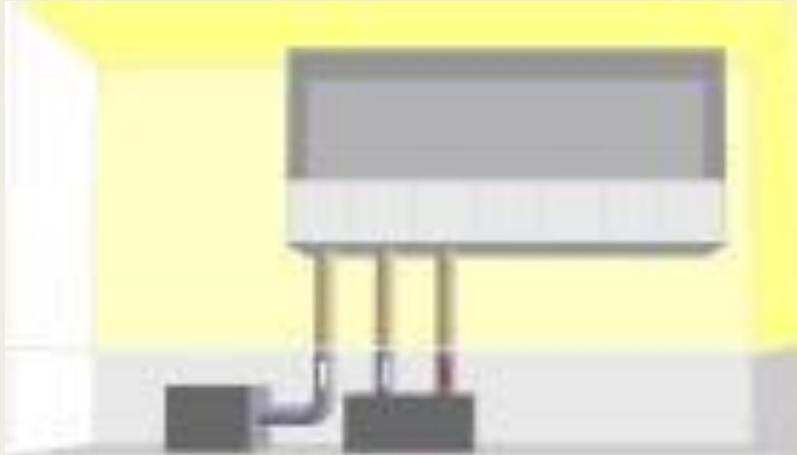
# Unix Pipes (aka Unnamed Pipes)

`int pipe(int fildes[2])`

- Returns a pair of file descriptors
  - *fildes[0] is connected to the read end of the pipe*
  - *fildes[1] is connected to the write end of the pipe*
- Create a message pipe
  - *Data is received in the order it was sent*
  - OS enforces mutual exclusion: only one process at a time
  - *Processes sharing the pipe must have same parent in common*

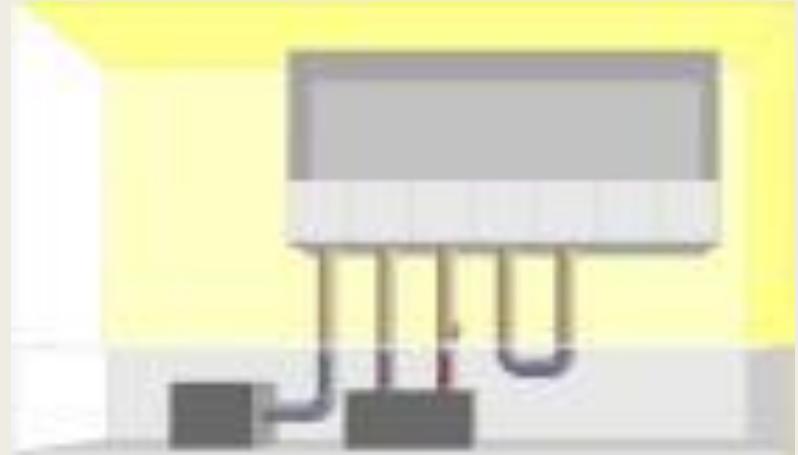
# Pipe Creation

BEFORE pipe



Process has some usual files open

AFTER pipe



Kernel creates a pipe and sets file descriptors

- BEFORE
  - Shows standard set of file descriptors
- AFTER
  - Shows newly created pipe in the kernel and the two connections to that pipe in the process

# IPC Pipe - Method

```
#include <stdio.h>
#include <unistd.h>

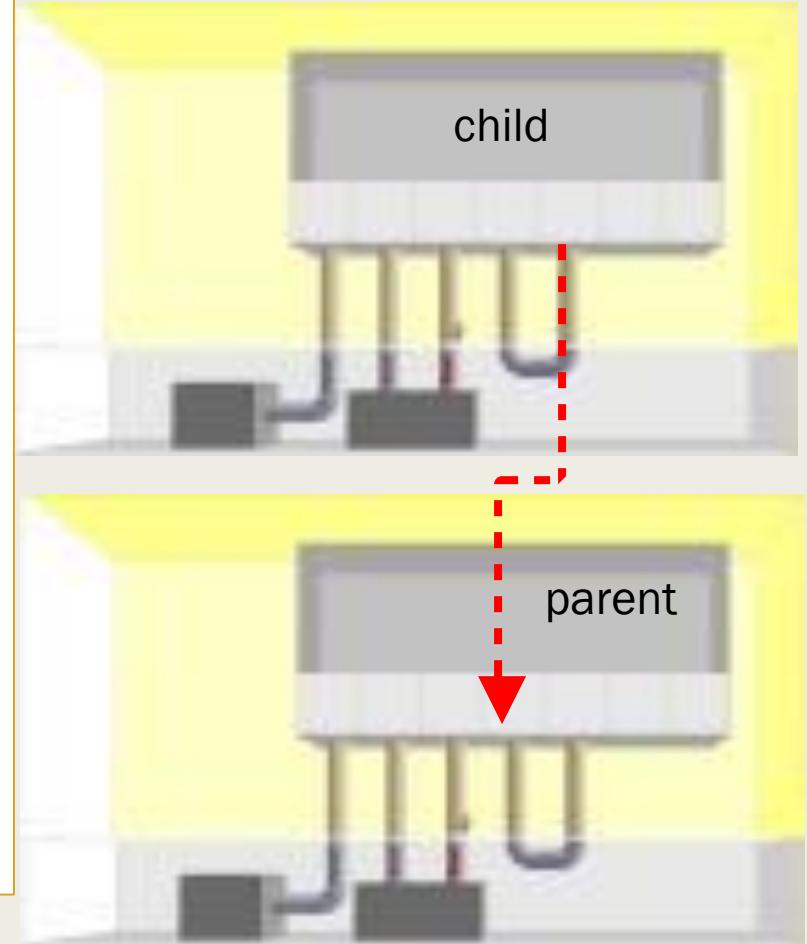
void main ()
{
    char buf [10];
    int fds [2];
    pipe (fds);
    printf ("sending msg: Hi\n");
    write (fds[1], "Hi", 3);
    read (fds[0], buf, 3);
    printf ("Received msg: %s\n", buf);
}
```

Connects the  
two fds as pipe

```
compute-Tinu1_tanzir/code> ./a.out
sending msg: Hi
Received msg: Hi
```

# Unnamed Pipe Between Two Processes

```
int main ()
{
    int fds [2];
    pipe (fds); // connect the pipe
    if (!fork()){ // on the child side
        sleep (3);
        char * msg = "a test message";
        printf ("CHILD: Sent %s\n", msg);
        write (fds [1], msg,
strlen(msg)+1);
    }else{
        char buf [100];
        read (fds [0], buf, 100);
        printf ("PRNT: Recvd %s\n", buf);
    }
    return 0;
}
```



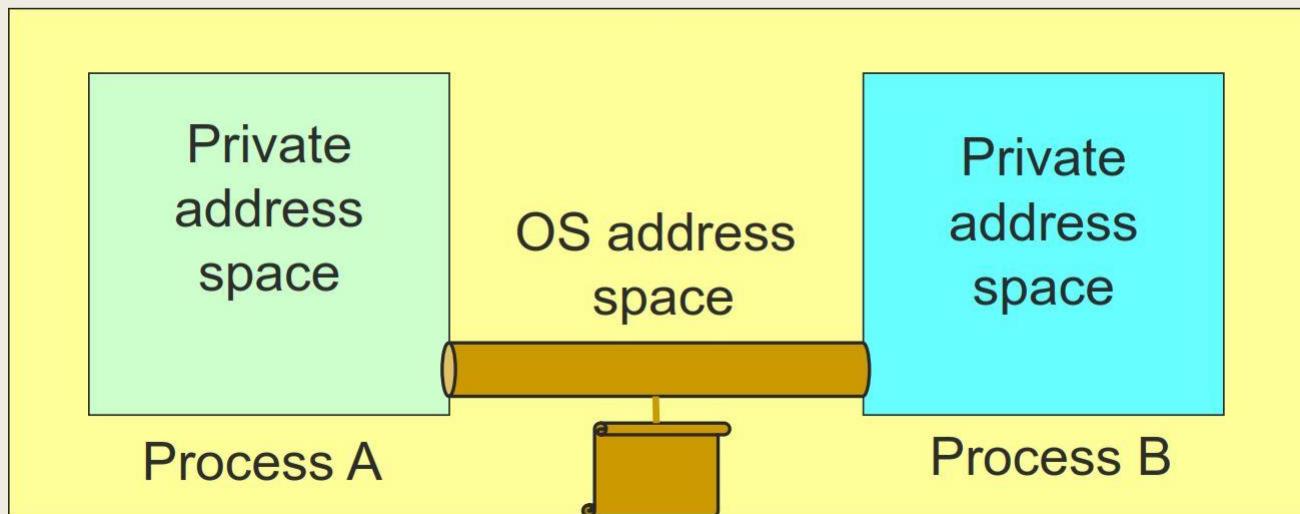
# Shell Piping: “ls -l | wc -l”

```
void main ()
{
    int fds [2];
    pipe (fds); // connect the pipe
    if (!fork()) { // on the child side
        close (fds[0]); // closing unnecessary pipe end
        dup2 (fds[1], 1); // overwriting stdout with pipeout
        execlp ("ls", "ls", "-l", NULL);
    } else {
        close (fds[1]); // closing unnecessary pipe end
        dup2 (fds[0], 0); // overwrite stdin with pipe in
        execlp ("wc", "wc", "-l", NULL);
    }
}
```

# IPC- FIFO (named PIPE)

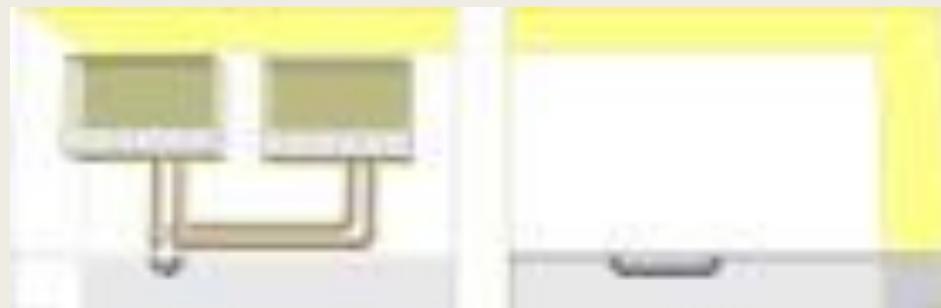
# FIFO

- A pipe (also called unnamed pipe) disappears when no process has it open
- FIFOs (also called named pipes) are a mechanism that allow for IPC that's like using regular files, except that the kernel takes care of synchronizing reads and writes, and
- Data is never actually written to disk (instead it is stored in buffers in memory) so the overhead of disk I/O (which is huge!) is avoided.



# FIFO vs PIPE

- A FIFO is like an unconnected garden hose lying on the lawn
  - *Anyone can put one end of the hose to his ear and another person can walk up to the hose and speak into the other end*
  - *Unrelated people may communicate through a hose*
  - *Hose exists even if nobody is using it*



# FIFO

- **It's part of the file system**

- *It has a name and path just like a regular file.*
  - *Programs can open it for reading and writing, just like a regular file.*
  - *However, the file does not contain any data*

- **Works like a Bounded Buffer**

- *Bytes travel in First-In-First-Out fashion: hence the name FIFO.*
  - *Reading process must wait for the writer*
  - *Writer must wait for the read operation once the buffer is full*

# FIFO - Problems

- We still need to agree on a name ahead of time – how to communicate that??

```
FIFORequestChannel rc ("control", ...){  
    ...  
    mkfifo ("control", PERMS); // create  
}
```

- Not concurrency safe within a process
  - *Like a file used by multiple processes/threads*
  - *Multiple threads writing can cause race condition*

# Using FIFO's

- How do I create a FIFO
  - *mkfifo (name)*
- How do I remove a FIFO
  - *rm fifoname or unlink(fifoname)*
- How do I listen at a FIFO for a connection
  - *open (fifoname, O\_RDONLY)*
- How do I open a FIFO in write mode?
  - *open(fifoname, O\_WRONLY)*
- How do two processes speak through a FIFO?
  - *The sending process uses write and the listening process uses read. When the writing process closes, the reader sees end of file*

# FIFO DEMO

## Writer

```
#define FIFO_NAME "test.txt"
int main(void)
{
    char s[300];
    int num, fd;
    mkfifo(FIFO_NAME, 0666); // create
    printf("Waiting for readers...\n");
    fd = open(FIFO_NAME, O_WRONLY); //open
    if (fd < 0)
        return 0;
    printf("Got a reader--type some
stuff\n");
    while (gets(s)) {
        if (!strcmp (s, "quit")) break;
        if ((num = write(fd, s, strlen(s))) == -1)
            perror("write");
        else
            printf("SENDER: wrote %d bytes\n",
num);
    }
    //unlink (FIFO_NAME);
    return 0;
}
```

## Reader

```
int main(void)
{
    char s[300];
    int num, fd;
    printf("waiting for writers...\n");
    fd = open(FIFO_NAME, O_RDONLY);
    printf("got a writer\n");
    do{
        if ((num = read(fd, s, 300)) == -1)
            perror("read");
        else {
            s[num] = '\0';
            printf("RECV: read %d bytes:
 \"%s\"\n", num, s);
        }
    } while (num > 0);
    return 0;
}
```

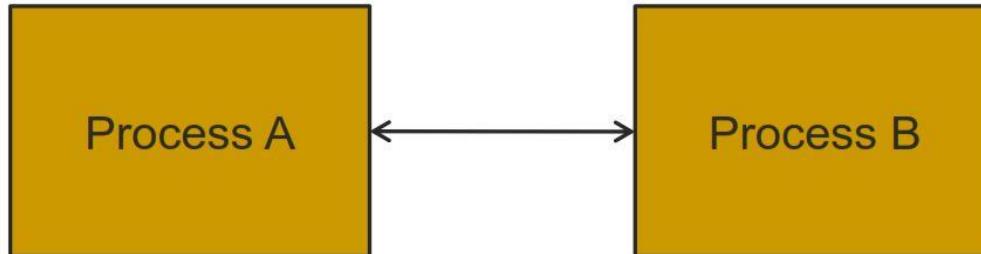
# IPC: Message Queue

# Message Queue

- Mechanism for processes to communicate and to synchronize their actions
- IPC facility provides two operations:
  - ***send(message)***
  - ***receive(message)***
- If  $P$  and  $Q$  wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive

# Message Passing

Direct



Indirect



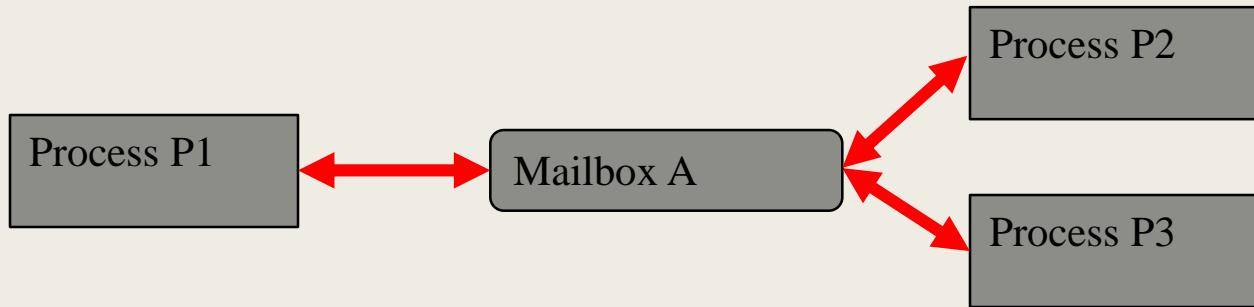
# Direct Message Passing

- Processes must name each other explicitly:
  - ***send*** ( $P$ , message) – send a message to process  $P$
  - ***receive***( $Q$ , message) – receive a message from process  $Q$
- Properties of communication link
  - Links are established automatically (or implicitly) while sending/receiving
  - A link is associated with exactly one pair of communicating processes
  - Between each pair, there exists exactly one link
- Limitation: Must know the name or id of the other process



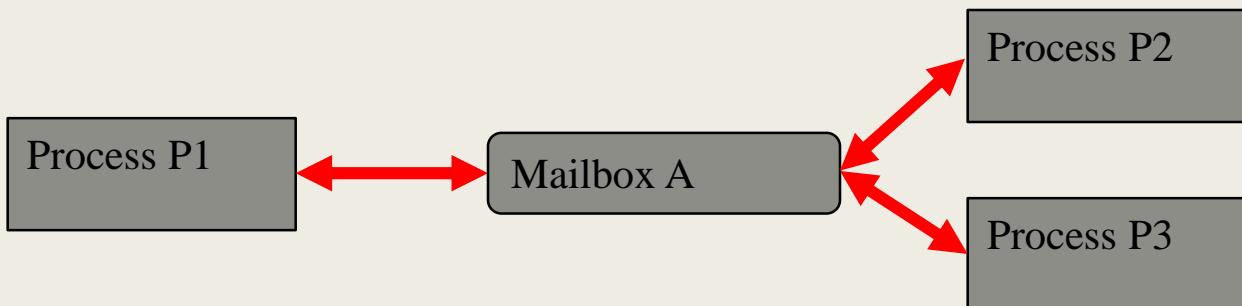
# Indirect Message Passing

- Messages are directed to and received from **mailboxes** (also referred to as ports)
  - *Mailbox can be owned by a process or by the OS*
  - *Each mailbox has a unique id*
  - *Processes can communicate only if they share a mailbox*
- Properties of communication link
  - *Link established only if processes share a common mailbox*
  - *A link may be associated with many processes*
  - *Each pair of processes may share several communication links*



# Indirect Message Passing

- Operations
  - *create a new mailbox*
  - *send and receive messages through mailbox*
  - *destroy a mailbox*
- Primitives are defined as:  
**send(A, message)** – send a message to mailbox A  
**receive(A, message)** – receive a message from mailbox A



# Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered **synchronous**
  - *Blocking send has the sender block until the message is received*
  - *Blocking receive has the receiver block until a message is available*
- Non-blocking is considered **asynchronous**
  - *Non-blocking send has the sender send the message and continue*
  - *Non-blocking receive has the receiver receive a valid message or null*
    - Does not wait for it
    - A receive may “fail” several times before it succeeds at the end

# Operations on Message Queues

```
mqd_t mq_open(const char *name, int oflag,  
               mode_t mode, struct mq_attr *attr);
```

```
int mq_send(mqd_t mqdes, const char *msg_ptr,  
            size_t msg_len, unsigned int msg_prio);
```

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,  
                   size_t msg_len, unsigned int *msg_prio)
```

```
int mq_close(mqd_t mqdes)
```

# Message Queue - Example

```
send() {
    mqd_t mq = mq_open("/testqueue", O_RDWR|O_CREAT, 0664, 0);
    if (mq_send(mq, av[1], strlen(av[1]) + 1, 0) < 0) {
        perror ("MQ Send failure");
        exit (0);
    }
    printf ("MQ Put: %s\n", av [1]);
    return 0;
}

receive() {
    mqd_t mq = mq_open("/testqueue", O_RDWR|O_CREAT, 0664, 0);
    struct mq_attr attr;
    mq_getattr (mq, &attr); // get attribute
    char *buf = (char*)malloc (attr.mq_msgsize);
    mq_receive(mq, buf, attr.mq_msgsize, NULL);
    printf ("MQ Receive Got: %s\n", buf);
    //clean-up
    mq_close(mq);
    //mq_unlink("/testqueue"); // remove from Kernel
    return 0;
}
```

# IPC: Shared Memory

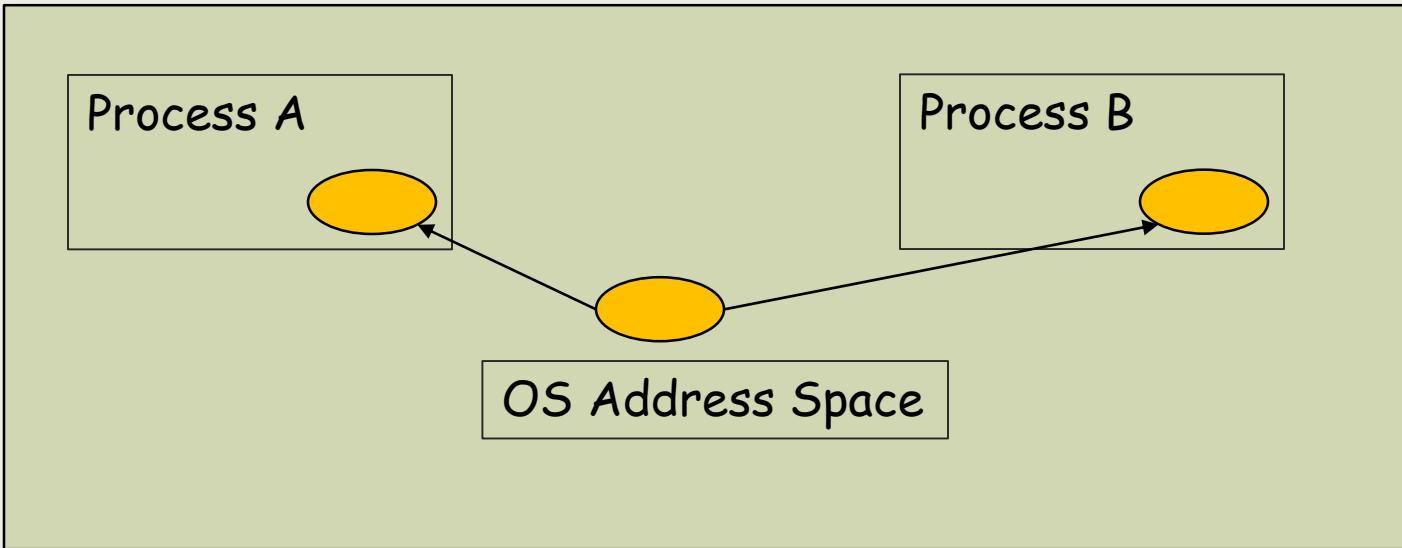
# Shared Memory

- How does data travel through a FIFO?
  - ‘write’ copies data from process memory to kernel buffer and then ‘read’ copies data from a kernel buffer to process memory
- If both processes are on the same machine, then they may not need to copy data in and out of the kernel
  - They may exchange or share data by using a shared memory segment
  - Shared memory is to processes what global variables are to threads

# Shared Memory

- Processes share the same segment of memory directly
  - *Memory is mapped into the address space of each sharing process*
  - *Memory is persistent beyond the lifetime of the creating or modifying processes (until deleted)*
- But, now the processes are on their own for synchronization
  - ***Mutual exclusion must be provided by processes using the shared memory***
  - ***Semaphores*** – to ensure Producer-Consumer relation also has to be now done by the respective processes

# Shared Memory



- Processes request the segment
- OS maintains the segment
- Processes can attach/detach the segment

# Facts about Shared Memory Segments

- A shared memory segment has a name, the name must start with “/”
- A shared memory segment has an owner and permission bits
- Processes may “map” or “unmap” a segment, obtaining/removing a pointer to the segment
- reads and writes to the memory segment are done via regular pointer operations

# Shared Memory – POSIX functions

- **shm\_open**: creates a shared memory segment
- **ftruncate**: sets the size of a shared memory segment
- **mmap**: maps the shared memory object to the process's address space
- **munmap**: unmaps from process's address space
- **shm\_unlink**: removes the shared memory segment from the kernel
- **close**: closes the file descriptor associated with the shared memory segment

# Shared Memory Example

```
char* my_shm_connect(char* name, int len){
    int fd = shm_open(name, O_RDWR|O_CREAT, 0644 );
    ftruncate(fd, len); //set the length to 1024, the default
is 0, so this is a necessary step
    char *ptr = (char *) mmap(NULL, len, PROT_READ|PROT_WRITE,
MAP_SHARED, fd, 0); // map
    if (fd < 0){
        perror ("Cannot create shared memory\n");
        exit (0);
    }
    return ptr;
}

void send(char* message){
    char *name    = "/testing";
    int len = 1024;
    char* ptr = my_shm_connect (name, len);

    strcpy(ptr, message); // putting data by just copying
    printf ("Put message: %s\n", message);
    close(fd); // close desc, does not remove the segment
    munmap (ptr, len); // this is a bit redundant,
}
```

# Shared Memory Example- contd

```
void receive() {
    char *name = "/testing";
    int len = 1024;
    char* ptr = my_shm_connect (name, len)

    printf ("Got message: %s\n", ptr);
    shm_unlink (name); //this removes the segment
from Kernel, this is a necessary clean up
    exit(0);
}
```

# Shared Memory - Summary

- It's great because it does not allocate extra memory
  - *3-times less memory (In others you have 1 copy in kernel, 2 others in the individual processes)*
- But a huge problem looms
  - Who does synchronization?
  - *There is no guarantee of order between sending and receiving processes*
  - *The receiver process can run first finding nothing or stale data in the buffer*

# Kernel Semaphores

- We have learned how to synchronize multiple threads using Semaphores and Locks
- But how do we synchronize multiple processes?
  - *We will again need semaphores, but this time Kernel Semaphores*
  - *They are visible to separate processes who do not share address space*

# Kernel Semaphore Operations

- **sem\_open(name, ...)** to create or connect to a semaphore
  - *The name argument must start with a “/”*
- **sem\_close()** closes a semaphore
  - *It does not destroy it from Kernel*
- **sem\_unlink()** removes from Kernel
  - *Must be put in the destructor for PA5*
- **sem\_wait()** is equivalent to Semaphore::P() operation
- **semd\_post()** is equivalent to Semaphore::V() operation
  
- Find out more from **sem\_overview(7)** in man pages

# COMPUTER NETWORKS

Tanzir Ahmed  
CSCE 313 Spring 2020

# Acknowledgments

- TAMU CSCE 313 Lecture Notes in Networking
  - *Acknowledgment: Profs Gu, Bettati, Tyagi*
- RICE COMP 221 Lecture Notes in Networking
  - *Acknowledgment: Prof. Cox*
- U-Illinois CS241 Lecture Notes in Networking
  - *Acknowledgment: Prof. Angrave*
- Beej's Guide to Network Programming, Ver. 3.0.15
- Socket Programming 101
  - *Acknowledgment: Vivek Ramachandran*
- U-Wisconsin CS-354 Lecture Notes in Networking
  - *Acknowledgment: Prof. Arpaci-Dusseau*

# Objectives

- Exposure to the basic underpinnings of the Internet
- Use network socket interfaces effectively

# The 2004 A. M. Turing Award



Bob Kahn

Vint Cerf

***"For pioneering work on internetworking, including the design and implementation of the Internet's basic communications protocols, TCP/IP, and for inspired leadership in networking."***

**The only Turing Award given to-date to recognize work in computer networking**

# But at the Same Time...

Daily, e.g. 110 attacks from China; 26 attacks from USA

2/2016 Hackers dumped the records of nearly 30,000 FBI and Department of Homeland Security workers.

1/2017 Hacking/Surveillance company Cellebrite lost 900GB of user data to a hack

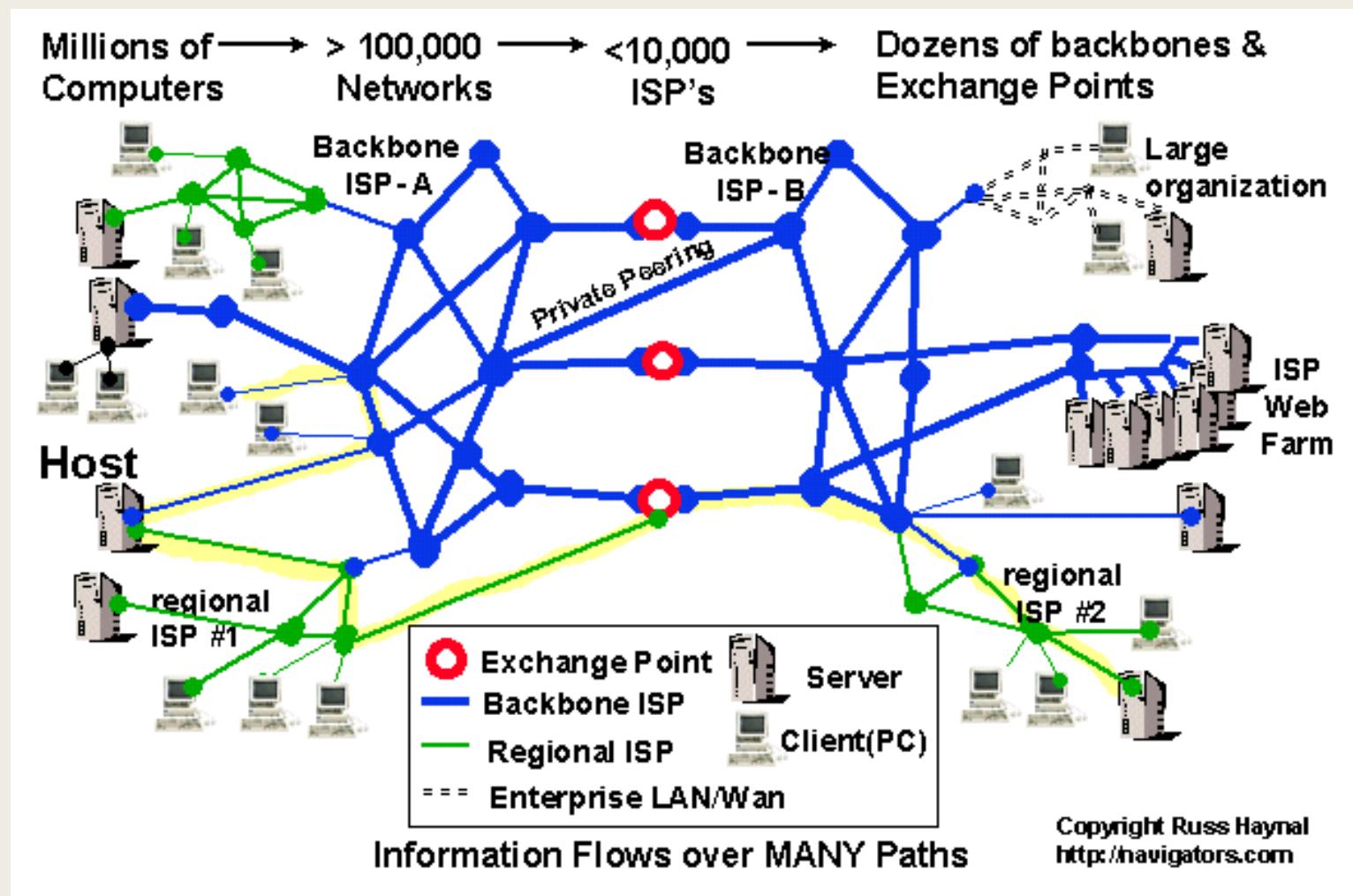
9/2018 Facebook security breach exposes personal data of 50M users

Source: Akamai Technologies, Inc.

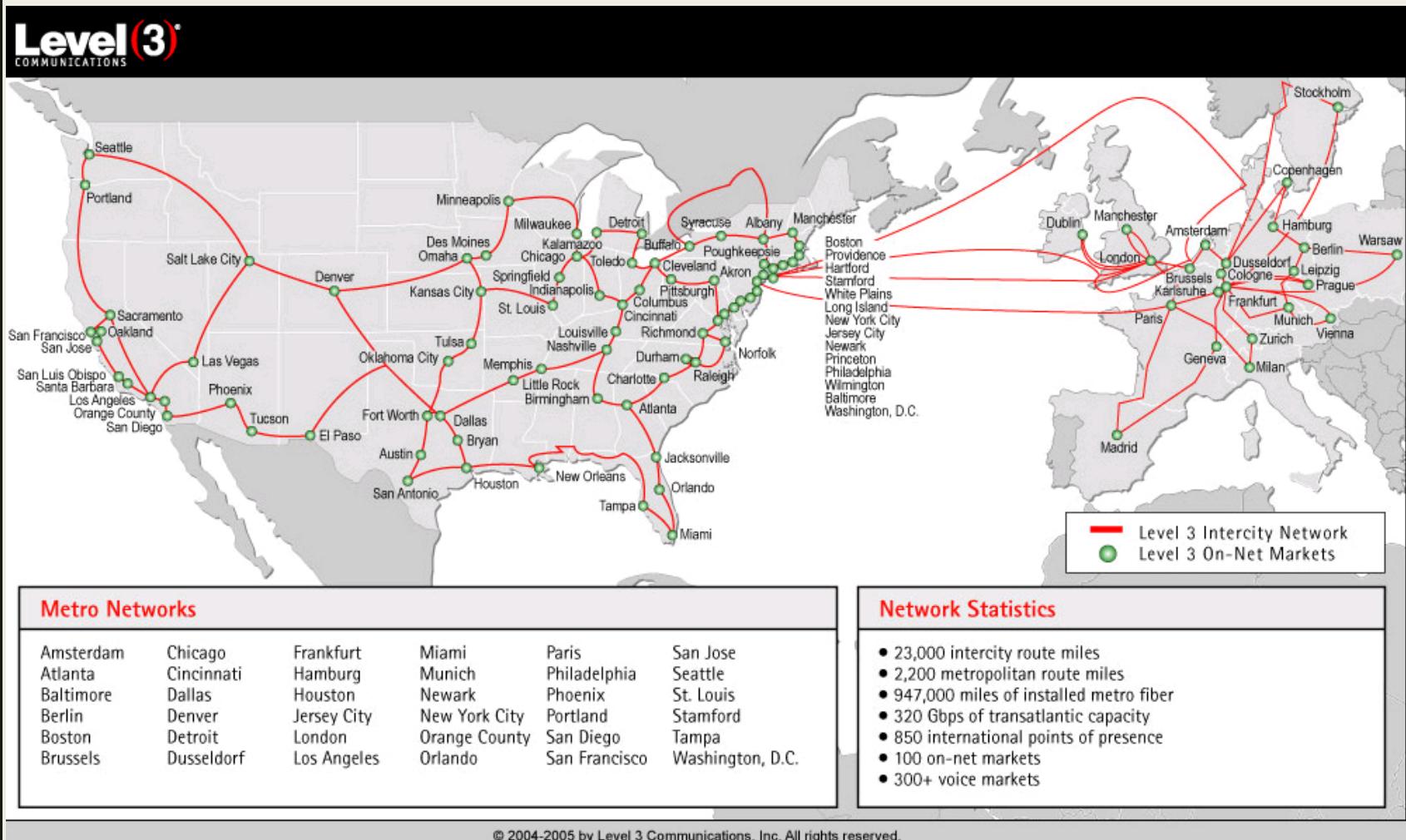
# Bottom-Line.....

- Internet is a ubiquitous presence in our lives
- Issues such as Security Lapses present themselves as **opportunities** for making our ways of communication more robust
- Let's now trace back the history from the early days of telephony in the next few slides

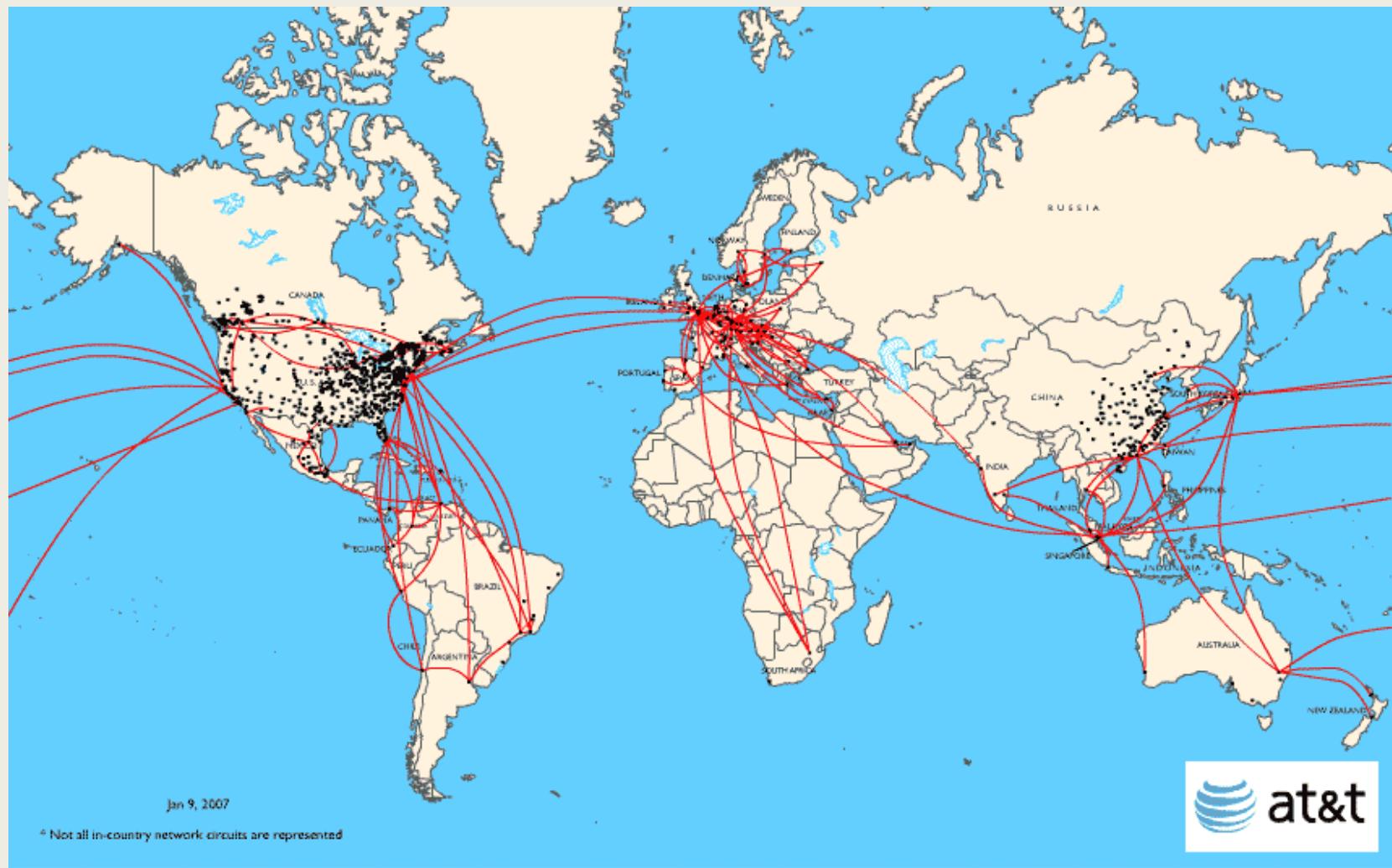
# Current Internet Architecture - Conceptual



# Level 3 Backbone



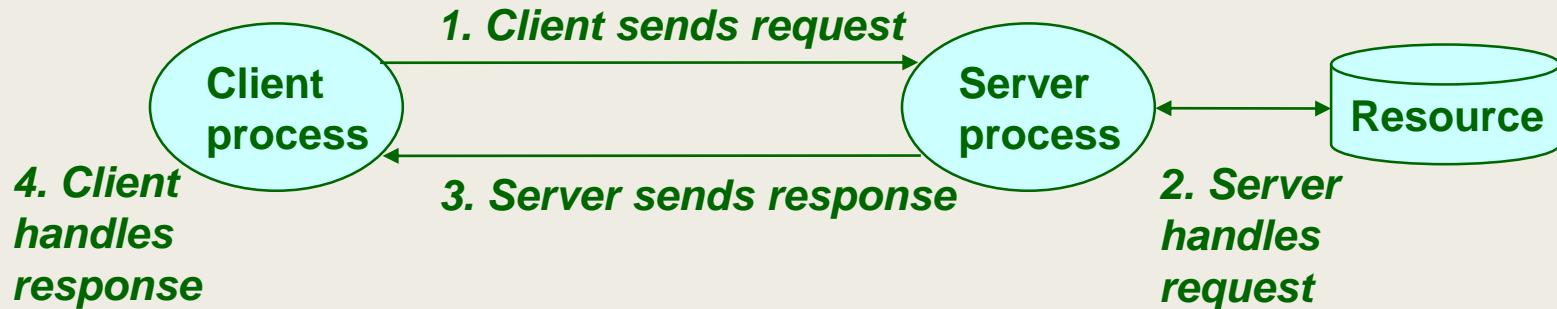
# AT&T Backbone



# A Client-Server Transaction

**Most network applications are based on the client-server model:**

- A server process and one or more client processes
- Server manages some resource
- Server provides service by manipulating resource for clients



*Note: clients and servers are processes running on hosts  
(can be the same or different hosts)*

# Computer Networks

A network is a hierarchical system of boxes and wires organized by geographical proximity

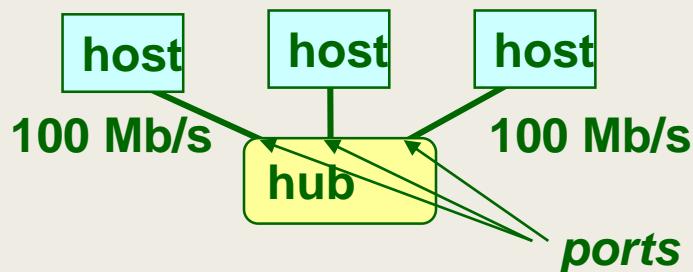
- Cluster network spans cluster or machine room
  - Switched Ethernet, Infiniband, ...
- LAN (local area network) spans a building or campus
  - Ethernet is most prominent example
- WAN (wide-area network) spans very long distance
  - A high-speed point-to-point link
  - Leased line or SONET/SDH circuit, or MPLS/ATM circuit

An internetwork (internet) is an interconnected set of networks

- The Global IP Internet (uppercase “I”) is the most famous example of an internet (lowercase “i”)

# Lowest Level of Connectivity: Ethernet Segment

Ethernet segment consists of a collection of hosts connected by wires (twisted pairs) to a hub



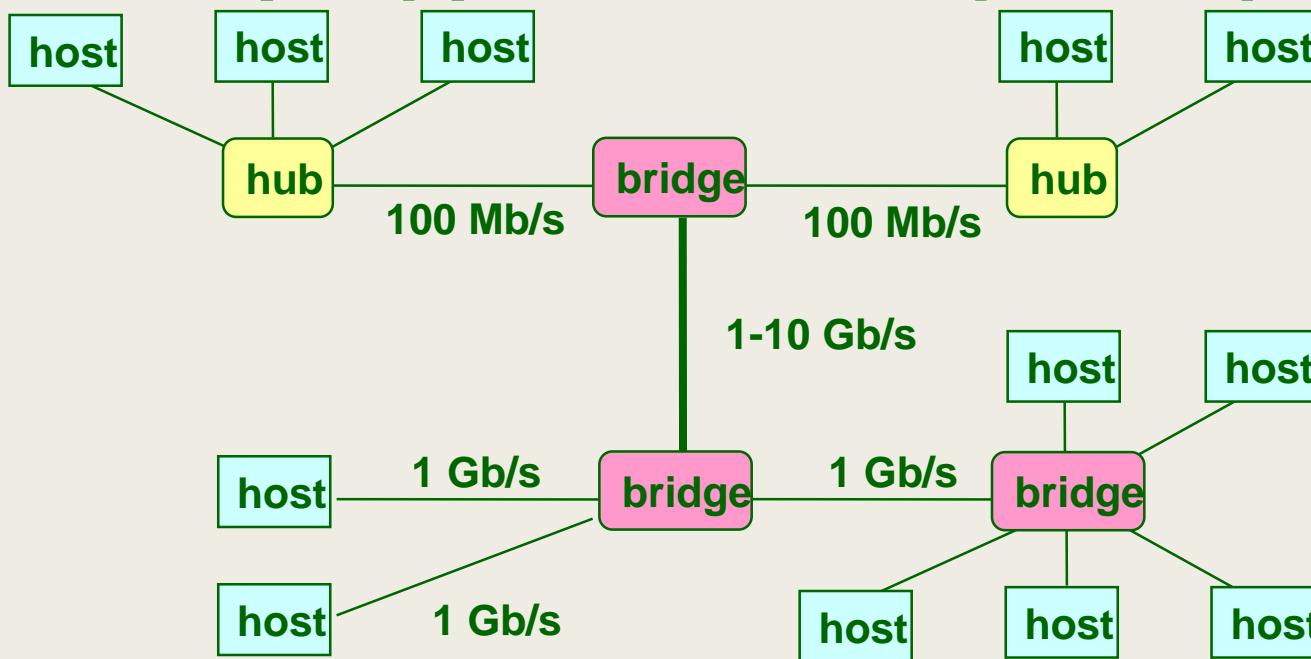
## Operation

- Each Ethernet adapter has a unique 48-bit address
- Hosts send bits to any other host in chunks called frames
- Hub copies each bit from each port to every other port
  - Every host sees every bit
- Note: Hubs are largely obsolete
  - Bridges (switches, routers) became cheap enough to replace them (don't broadcast all traffic)

# Next Level: Bridged Ethernet Segment

**Spans room, building, or campus**

**Bridges cleverly learn which hosts are  
reachable from which ports and then  
selectively copy frames from port to port**



# Conceptual View of LANs

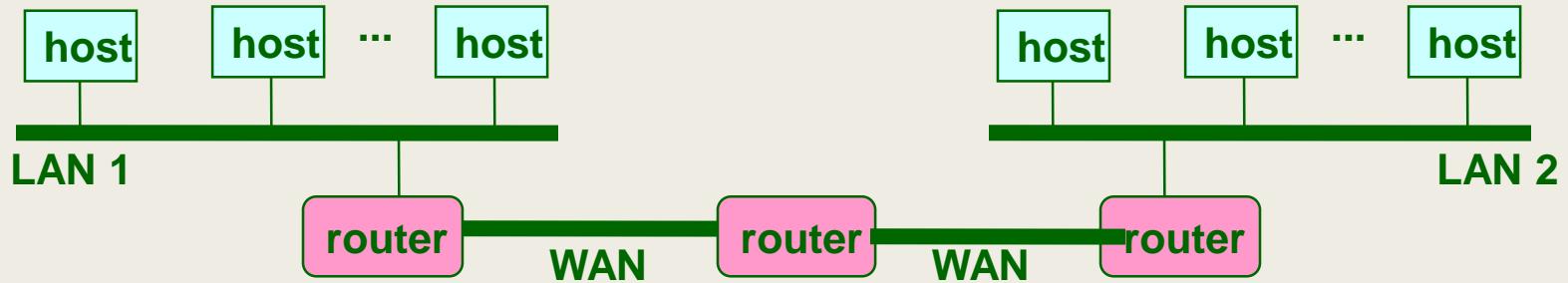
**For simplicity, hubs, bridges, and wires are often shown as a collection of hosts attached to a single wire:**



# Next Level: internets

Multiple **incompatible LANs** can be physically connected by specialized computers called **Routers**

The connected networks are called an **internet**



LAN 1 and LAN 2 might be completely different, totally incompatible LANs (e.g., Ethernet and WiFi, 802.11\*, T1-links, DSL, ...)

# The Notion of an Internet Protocol

**How is it possible to send bits across incompatible LANs and WANs?**

**Solution: Protocol software running on each host and router smoothens out the differences between the different networks**

**Implements an internet protocol (i.e., set of rules) that governs how hosts and routers should cooperate when they transfer data from network to network**

- ♦ **TCP/IP is the protocol for the global IP Internet**

# What Does an Internet Protocol Do?

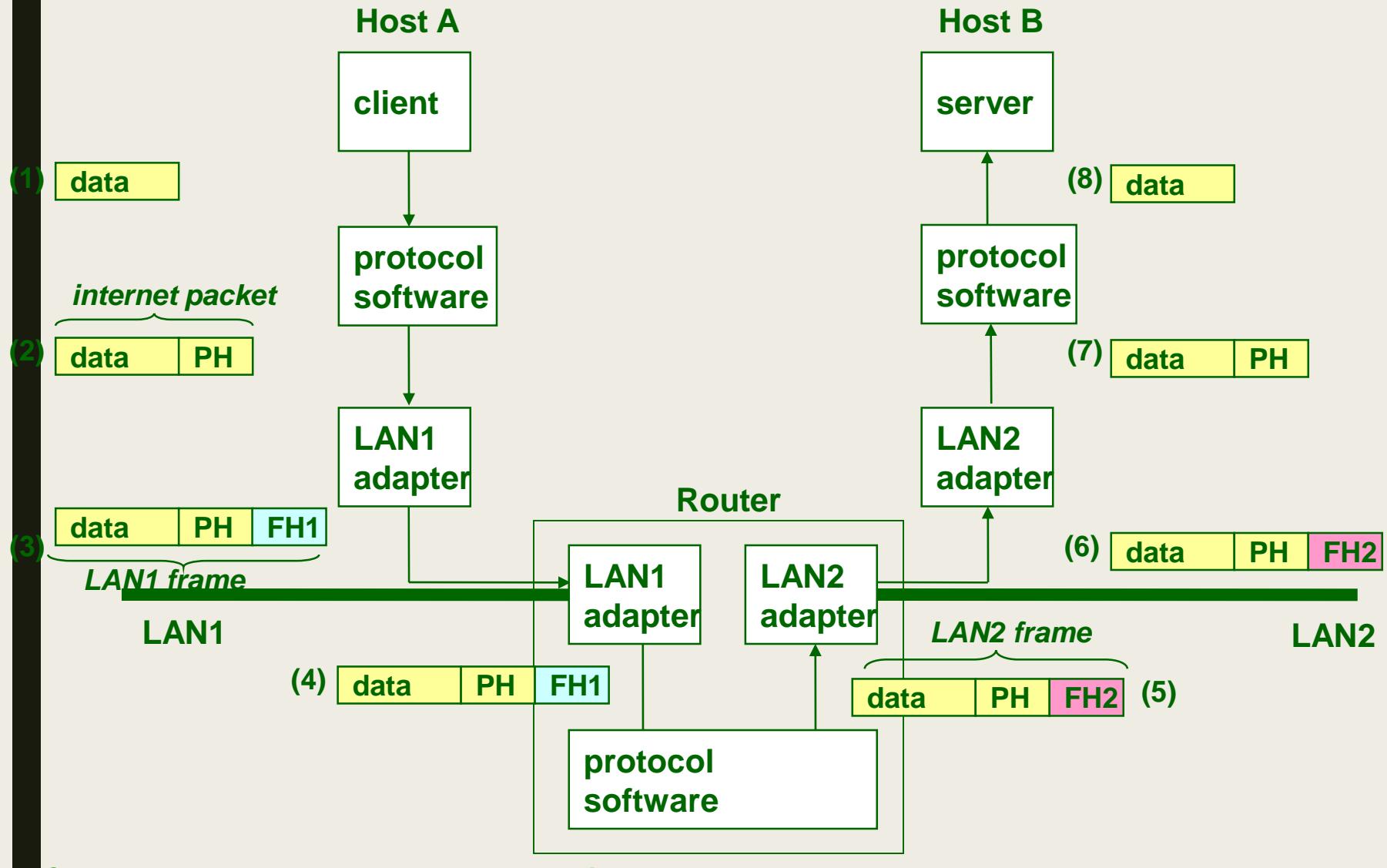
## 1. Provides a naming scheme

- An internet protocol defines a uniform format for host addresses
- Each host (and router) is assigned at least one of these internet addresses that uniquely identifies it

## 2. Provides a delivery mechanism

- An internet protocol defines a standard transfer unit (packet)
- Packet consists of header and payload
  - Header: contains info such as packet size, source and destination addresses
  - Payload: contains data bits sent from source host

# Transferring Data Over an Internet



# Other Issues

**We are glossing over a number of important questions:**

- **What if different networks have different maximum frame sizes? (segmentation)**
- **How do routers know where to forward frames?**
- **How are routers informed when the network topology changes?**
- **What if packets get lost?**

**We'll leave the discussion of these question to computer networking classes**

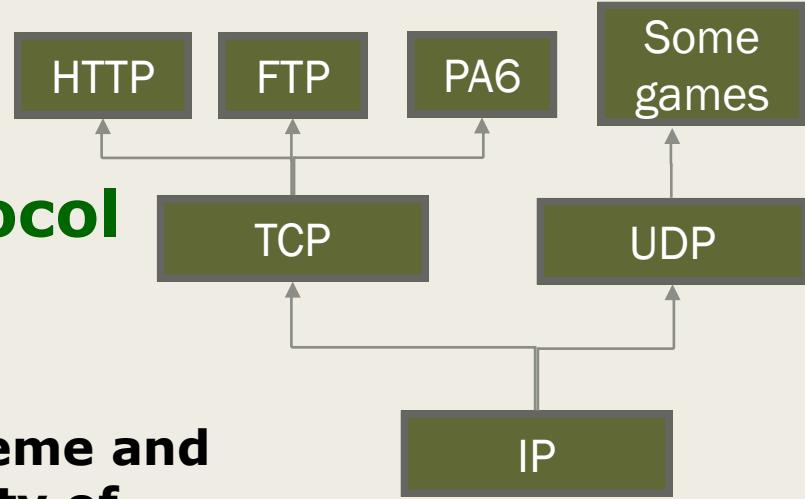
→ CSCE 463

# Global IP Internet

Based on the TCP/IP protocol family

- ◆ **IP (Internet protocol) :**
  - Provides basic naming scheme and unreliable delivery capability of packets (datagrams) from host-to-host
- ◆ **UDP (User Datagram Protocol)**
  - Uses IP to provide **unreliable** datagram delivery from process-to-process
- ◆ **TCP (Transmission Control Protocol)**
  - Uses IP to provide **reliable** byte streams from process-to-process over connections

Accessed via a mix of Unix file I/O and functions from the sockets interface



# A Programmer's View of the Internet

## Hosts are mapped to a set of 32-bit IP addresses

- e.g. **128.194.255.88 (4 \* 8 bits)**

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian) */
};
```

## A set of identifiers called Internet domain names are mapped to the set of IP addresses for convenience (Domain Name Server aka DNS)

- **linux2.cs.tamu.edu** is mapped to **128.194.138.88**
- **A process on one Internet host can communicate with a process on another Internet host over a connection**

# Dotted Decimal Notation

**By convention, each byte in a 32-bit IP address is represented by its decimal value and separated by a period**

- IP address 0x8002C2F2 = 128.2.194.242

**Functions for converting between binary IP addresses and dotted decimal strings:**

- `inet_ntop`: converts a dotted decimal string to an IP address in network byte order
- `inet_ntop`: converts an IP address in network byte order to its corresponding dotted decimal string
- “n” denotes network representation, “p” denotes presentation representation

# IP Address Structure

**IP (V4) Address space divided into classes:**

	0	1	2	3	8	16	24	31
Class A	0	Net ID				Host ID		
Class B	1	0	Net ID			Host ID		
Class C	1	1	0	Net ID			Host ID	
Class D	1	1	1	0	Multicast address			
Class E	1	1	1	1	Reserved for experiments			

**Special Addresses for routers and gateways  
(all 0/1's)**

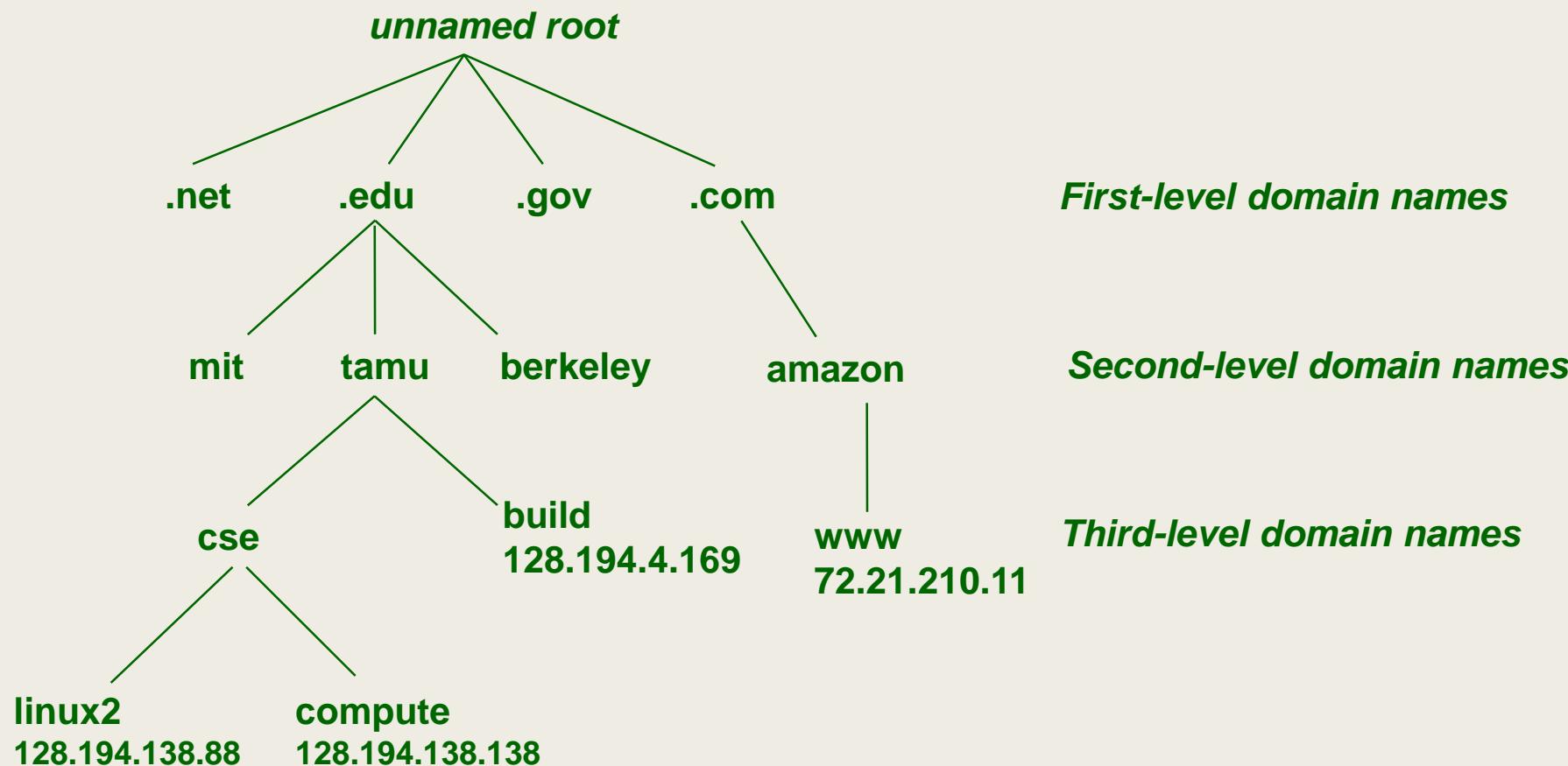
**Loop-back address: 127.0.0.1**

**Un-routed (private) IP addresses:**

- 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16

**Dynamic IP addresses (DHCP)**

# Internet Domain Names



# Domain Naming System (DNS)

**The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called DNS**

- Conceptually, programmers can view the DNS database as a collection of millions of addrinfo structures:

**DNS has uses beyond just IP address lookup:**

- Load balancing for busy servers (e.g., google.com)
- Finding the nearest site for content (e.g., youtube.com)

**Functions for retrieving host entries from DNS:**

- `getaddrinfo`: query DNS using domain name or IP

# Querying DNS

**Domain Information Groper (dig) provides a scriptable command line interface to DNS**

```
unix> dig +short linux2.cse.tamu.edu  
128.194.138.88  
unix> dig +short -x 128.194.138.85  
chevron.cs.tamu.edu.  
unix> dig +short www.google.com  
216.58.194.36  
unix> dig +short www.google.com  
172.217.12.68  
unix> dig +short build.tamu.edu  
compute.cse.tamu.edu.  
128.194.138.139
```

Shows just IP

Reverse lookup

Same server,  
different IPs

Shows  
aliases

# Internet Connections

**Clients and servers communicate by sending streams of bytes over connections:**

- Point-to-point, full-duplex (2-way communication), and reliable

**A socket is an endpoint of a connection**

- Socket address is an IP address and port pair

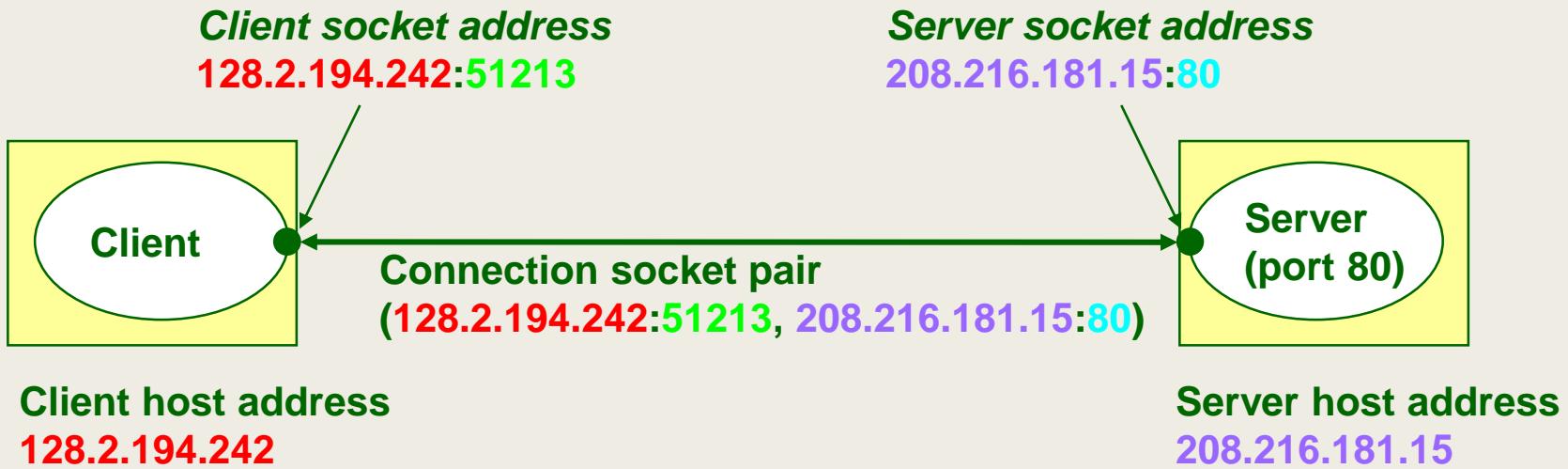
**A port is a 16-bit integer that identifies a process:**

- Ephemeral port: Assigned automatically on client when client makes a connection request
- Well-known port: Associated with some service provided by a server (e.g., port 80 is associated with Web servers)

**A connection is uniquely identified by the socket addresses of its endpoints (socket pair)**

- (cliaddr:cliport, servaddr:servport)

# Putting it all Together: Anatomy of an Internet Connection



# Prerequisites for an Internet Connection

- There must be a network path between the Client and the Server
  - For instance, there is a path between your cell phone and your computer residing in the same WiFi network
  - A path between your desktop and google.com
- **Firewalls** could artificially **block** network paths
  - Path between off-campus computer and build.tamu.edu is blocked by sys admins
  - Path between compute.cs and linux2.cs is also blocked by admins
  - Amazon EC2 servers do not accept outside connections except for specific ports

# Testing for Network Paths

## ping command

- Tests if you can reach another host
- Example 1: From off-campus (w/o VPN)

```
osboxes@osboxes:~$ ping www.google.com
PING www.google.com (172.217.1.132) 56(84) bytes of data.
64 bytes from atl14s07-in-f132.1e100.net (172.217.1.132): icmp_seq=1 ttl=55 time=13.8 ms
64 bytes from atl14s07-in-f132.1e100.net (172.217.1.132): icmp_seq=2 ttl=55 time=14.9 ms
64 bytes from atl14s07-in-f132.1e100.net (172.217.1.132): icmp_seq=3 ttl=55 time=13.3 ms
64 bytes from atl14s07-in-f132.1e100.net (172.217.1.132): icmp_seq=4 ttl=55 time=13.5 ms
^C
--- www.google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3009ms
rtt min/avg/max/mdev = 13.296/13.863/14.858/0.599 ms
osboxes@osboxes:~$ ping build.tamu.edu
PING compute.cse.tamu.edu (128.194.138.139) 56(84) bytes of data.
^C
--- compute.cse.tamu.edu ping statistics ---
9 packets transmitted, 0 received, 100% packet loss, time 8195ms

osboxes@osboxes:~$ 
```

- Example 2: Now with VPN connected

```
osboxes@osboxes:~$ ping build.tamu.edu
PING compute.cse.tamu.edu (128.194.138.139) 56(84) bytes of data.
64 bytes from compute.cs.tamu.edu (128.194.138.139): icmp_seq=1 ttl=59 time=23.2 ms
64 bytes from compute.cs.tamu.edu (128.194.138.139): icmp_seq=2 ttl=59 time=23.7 ms
64 bytes from compute.cs.tamu.edu (128.194.138.139): icmp_seq=3 ttl=59 time=23.0 ms
64 bytes from compute.cs.tamu.edu (128.194.138.139): icmp_seq=4 ttl=59 time=24.4 ms
^C
--- compute.cse.tamu.edu ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3006ms
rtt min/avg/max/mdev = 22.999/23.560/24.391/0.544 ms
```

# TCP Connectivity in the Path

## telent command

- A built in TCP client, used to check server existence or connection ability
- Tests if you can make a TCP connection to a server port
- Example 1: Connecting to port 80 (HTTP server) of google

```
osboxes@osboxes:~$ telnet www.google.com 80
Trying 172.217.14.164...
Connected to www.google.com.
Escape character is '^]'.
```

- Example 2: Connecting to port 80 of build.tamu

```
osboxes@osboxes:~$ telnet build.tamu.edu 80
Trying 128.194.138.139...
^C
osboxes@osboxes:~$
```

- Connection not possible because:
  - No service at port 80
  - Even if it had, firewall does not allow

# Clients

## Examples of client programs

- Web browsers, ftp, telnet, ssh

## How does a client find the server?

- The IP address in the server socket address identifies the host (more precisely, an adapter on the host)
- The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service

# Servers

## **Servers are long-running processes (daemons)**

- **Created at boot-time (typically) by the init process (process 1)**
- **Run continuously until the machine is turned off**

**Each server waits for requests to arrive on a well-known port associated with a particular service**

- **Port 23: telnet server**
- **Port 25: mail server**
- **Port 80: HTTP server**

**A machine that runs a server process is also often referred to as a “server”**

# Server Examples

## Web server (port 80)

- **HTML files/content/other data through HTTP protocol**

## FTP server (20, 21)

- **Service: stores and retrieve files**

## Telnet server (23)

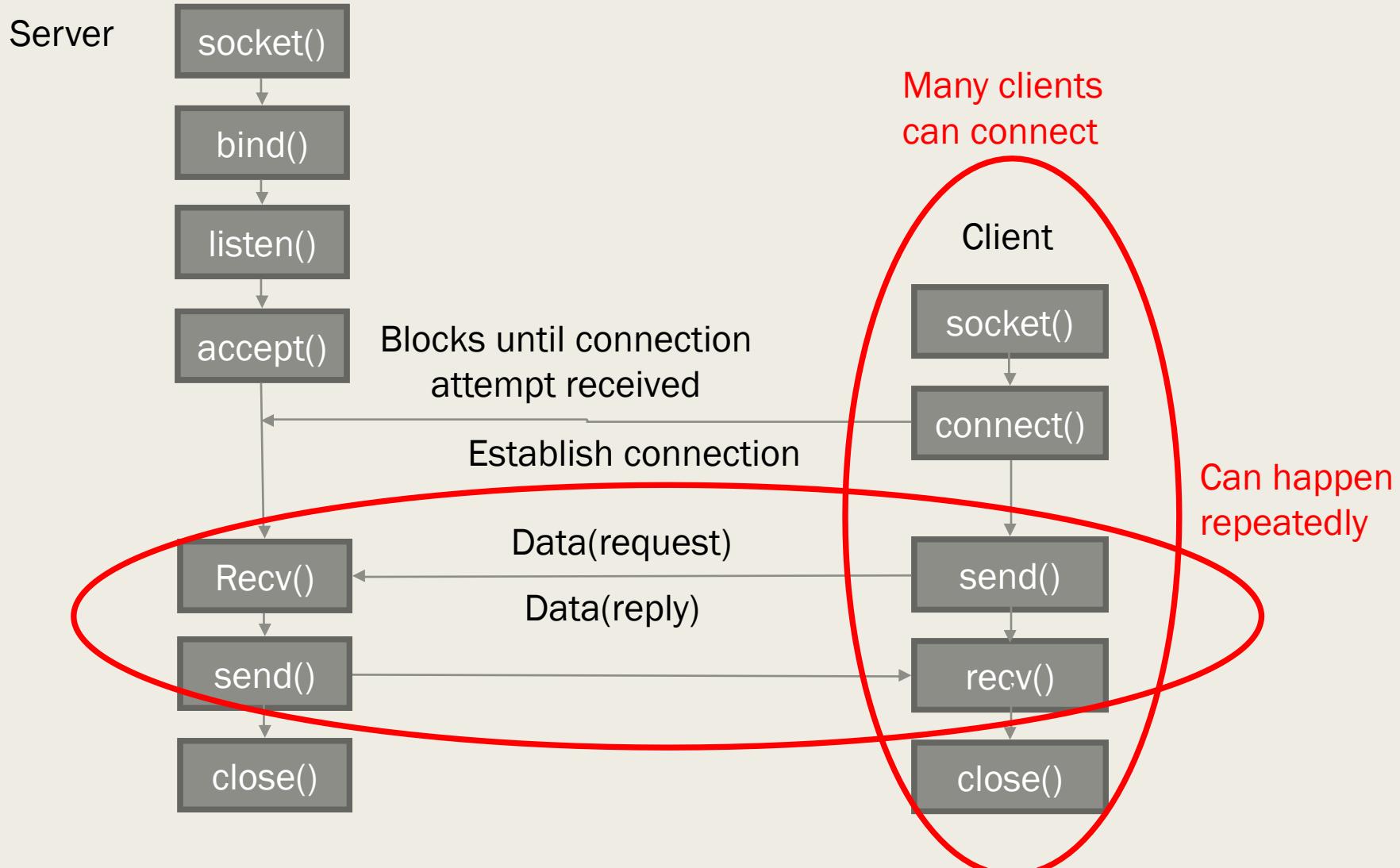
- **Resource: terminal**
- **Service: proxies a terminal on the server machine**

## Mail server (25)

- **Resource: email “spool” file**
- **Service: stores mail messages in spool file**

See `/etc/services` for a comprehensive list of the services available on a UNIX machine

# A Server-Client Interaction in TCP – POSIX Functions



# A Closer Look into POSIX Functions

- `getaddrinfo()`
- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `connect()`
- `write()` , `send()` , `sendto()`
- `read()` , `recv()` , `recvfrom()`
- `close()`

# Data Structures

```
/* structure for looking up IP address */
struct addrinfo{
    int      ai_flags;
    int      ai_family; // AF_INET=IPv4,AF_INET6= IPv6
    int      ai_socktype;        // TCP or UDP
    int      ai_protocol;
    socklen_t ai_addrlen;      // length of ai_addr
    struct sockaddr* ai_addr; // contains IP+PORT
    char*   ai_canonname;       // canonical name
    struct addrinfo* ai_next; // next pointer of result
link list
};

/* data structure for IP details (+PORT) */
struct sockaddr_in{
    short      sin_family; // IPv4 or IPv6
    unsigned short sin_port; // port number
    struct in_addr sin_addr; // 32 bit IP
    char sin_zero [8];
};

/* just a wrapper for the numeric IP */
struct in_addr{
    unsigned long s_addr;
};
```

# **getaddrinfo() – Looking Up IP address from name**

- The first step to locate a server by the client
- Converts easy-to-remember DNS names (e.g., `linux.cs.tamu.edu`) into machine-usable IP address
- Queries DNS servers (a collection of mappings)

```
int getaddrinfo(char* name, char* port, struct addrinfo* hints, struct addrinfo** result);
```

`name` = name of the host

`port` = the port where the service (e.g., http, your data server in MP6) is running

`hints` = provides some initial hint (IPv4/IPv6, TCP/UDP etc.)

`result` = linked list of looked up addresses

- **Example:**

```
getaddrinfo("www.example.com", "3490", &hints, &res);
```

# getaddrinfo() - Detailed

```
int status;
struct addrinfo hints;
struct addrinfo *servinfo; // will point to the results

//preparing hints data structure
memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_UNSPEC; // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets

// look up the IP address from the name: "www.example.com"
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);

for(p = res;p != NULL; p = p->ai_next) {
    void *addr;
    char *ipver;
    // get the pointer to the address itself,
    // different fields in IPv4 and IPv6:
    if (p->ai_family == AF_INET) { // IPv4
        struct sockaddr_in *ipv4 = (struct sockaddr_in *)p->ai_addr;
        addr = &(ipv4->sin_addr);
        ipver = "IPv4";
    } else { // IPv6
        struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)p->ai_addr;
        addr = &(ipv6->sin6_addr);
        ipver = "IPv6";
    }
    // convert the IP to a string and print it:
    inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
    printf("%s: %s\n", ipver, ipstr);
}
```

# **socket() – A Connection End Point**

- **Creates a communication end-point for a network connection**

```
int socket (int domain, int type, int protocol)  
domain = PF_INET (IPv4) / PF_INET6 (IPv6)  
type = SOCK_STREAM (TCP) / SOCK_DGRAM (UDP)  
protocol = 0
```

- **Example:**

```
s = socket (AF_INET, SOCK_STREAM, 0)
```

will create a TCP socket

- **The above call returns 0 on success and -1 on failure**

# connect() – Client Attempting Server Connection

- This is called by the client to attempt a connection with the server
- Blocks until the server accepts it

```
int connect(int sockfd, struct sockaddr* server,  
socketlen_t server_len)
```

Sockfd: socket variable prepared beforehand

server = address of the server (returned by  
getaddrinfo)

- Example:

```
getaddrinfo("www.example.com", "3490", &hints, &res); // lookup  
// make a socket:  
sockfd = socket(res->ai_family, res->ai_socktype, res-  
>ai_protocol);  
// connect to server. Once successful, the socket becomes ready as the endpoint  
connect(sockfd, res->ai_addr, res->ai_addrlen);
```

# **close() – Close a Session**

- **Called by both the client and the server**
- **Signals end of a communication**
- **Internally, frees resources associated with a connection**
  - **Important for busy servers, also for PA7**  
int close(int sock)
- **Example:**

```
close (sock)
```

# bind() – Server Attaching to a Port

- A server process calls this to associate its socket to a given port
- Port number is used by the kernel to forward an incoming packet to a certain process (its socket)

```
int bind(int sockfd, struct sockaddr* addr,  
socketlen_t addrlen)
```

- Example:

```
getaddrinfo(NULL, "3490", &hints, &res); // lookup  
// make a socket:  
sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);  
// bind it to the port we passed in to getaddrinfo():  
bind(sockfd, res->ai_addr, res->ai_addrlen);
```

# listen() – Setting up Server

- This is a prerequisite before a connection is accepted
- Incoming connections wait in a queue before accepted, listen () sets the size of that queue

```
int listen(int sockfd, int backlog)
```

- Example:

```
listen (sockfd, 20); // should be replaced by w in your PA6
```

# accept() – Server Accepting Client Connection

- **This is called by the server to accept a new client connection**

```
int accept(int sockfd, struct sockaddr* client,  
socketlen_t client_len)  
sockfd = socket  
client = will hold the client address details  
client_len = address length
```

- **Example:**

```
struct sockaddr client;  
accept (sockfd, &client, sizeof (client));
```

# send()/recv() – Finally Data

- **Called by both client and server to exchange data**
- **Blocks until the server accepts it**

```
int send(int sock, void* msg, size_t len, int flags)
int recv(int sock, void* msg, size_t len, int flags)
```

Msg = buffer pointer to send/receive data from/to

Len = sender: length of the message,

receiver: buffer capacity (to avoid overflow)

- **Example:**

```
char *send_msg = "a sample message";
int sent_bytes = send (sockfd, send_msg, strlen (send_msg)+1, 0);
```

```
char recv_buffer [1024];
int recv_len = recv (sockfd, recv_buffer, 1024, 0);
```

# FILE SYSTEMS

Tanzir Ahmed  
CSCE 313 Spring 2020

# The UNIX File System

- File Systems and Directories
- UNIX inodes
- File protection/access control

# Why Study File Systems?

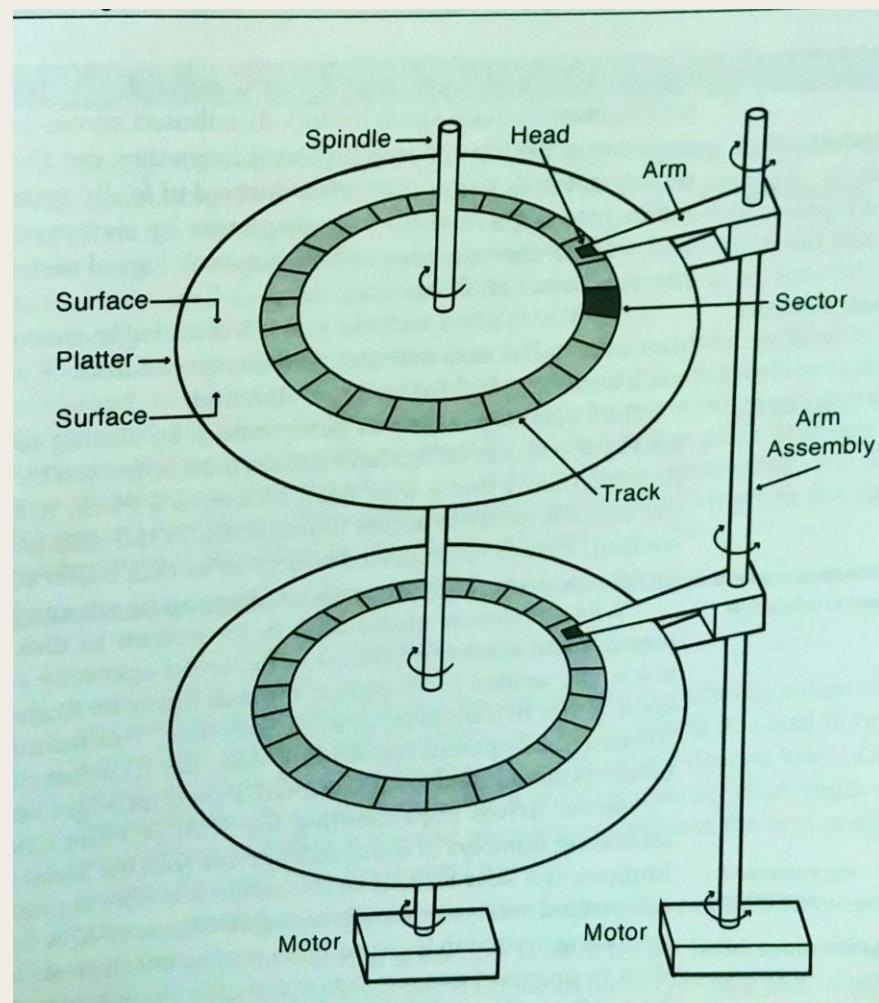
- The main subject is non-volatile storage, e.g.,
  - *Magnetic disk*
  - *Tape drives*
  - *Flash disks*
  - *SSDs*
- These retain data between shutdown and reboots
  - *Important for keeping the OS, programs, and user data*
- But these devices have unique characteristics and limitations not found in main memory
  - *First, they do not allow random accesses to individual bytes*
    - only allow block level access (512 bytes or multiple)
  - *Second, accesses are much slower* (10ms compared to 0.1ms of memory)

# Why Study File Systems?

- The physical characteristics drive the following keys features of file systems:
  - **Named data:** *Human-readable names (e.g. file names) given to data*
  - **Performance:** *By grouping, ordering, scheduling disk operations so that high latency is hidden/amortized*
  - **Reliability:** *Unexpected power-cycles do not corrupt the data*
  - **Controlled sharing:** *Determine who can read/write/execute certain files sequentially/simultaneously w/o corrupting*

# Now, Why Are Disks Slow?

- Consider a typical magnetic disk
  - *Platters are rotating at a constant speed*
  - *Each surface has data in the form of a number of tracks*
    - Each track is separated in sectors/blocks
  - *Arm moves the disk head to place that on the right track. This is called “seek”*
    - Extremely slow (> 10 ms)
  - *Then, the head waits for the correct sector to come below*
    - Relatively faster because of constant motion (<10 ms), determined by disk RPM
  - *Read the sector and send to the CPU*
    - Much faster (at SATA rate: ~500MB/s)



# Disk Access Time

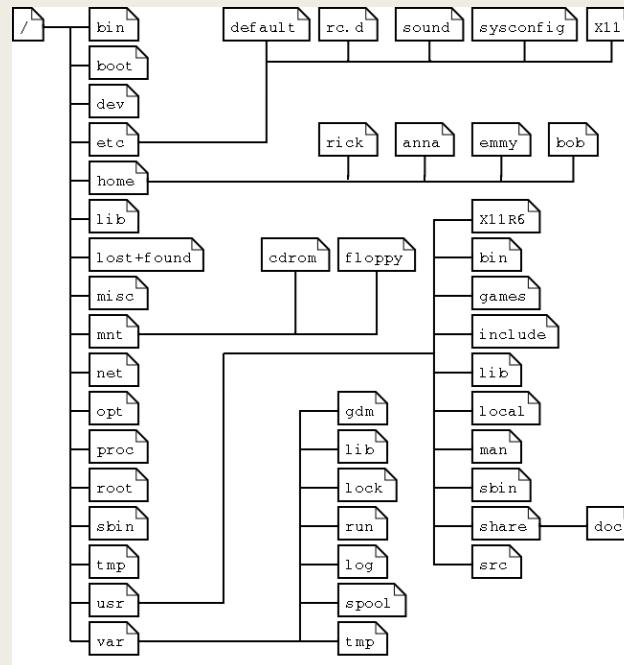
- disk access time = **seek time** + rotation time + transfer time
- The mechanical movement of the disk arm makes the seek time often the bottleneck
- Sequentially read => less disk seeking
  - *It takes less time to move between adjacent tracks*
- If data is read randomly, the disk arm has to seek randomly, leading to very slow operation
  - *Even then, it applies algorithms similar to the ones in elevators to service requests*

# How About SSDs?

- Are they “true random access” – meaning do they take same time as sequential even when accessed randomly?
  - *The answer is NO*
  - *Because accesses are still at sector or page level*
  - *Accessing just 1 byte experiences the full page transfer latency*
  - *In addition, prefetching an adjacent page is a common practice, which hides latency for sequential, but not for random*
  - *Overall, random access is faster than magnetic, but not as fast as sequential*
- From that same logic, even RAM is not truly random access
  - *You will always have a leverage accessing sequantially*

# File System Abstraction

- Definition: “*File system is an OS abstraction that provides persistent and named data*”
- Key components:
  - **Files:** contain **metadata** and **actual data**
  - **Directories:** Special files that contain **names** and **pointers** to actual files
- Because of directories and sub-directories, the File System looks like a tree

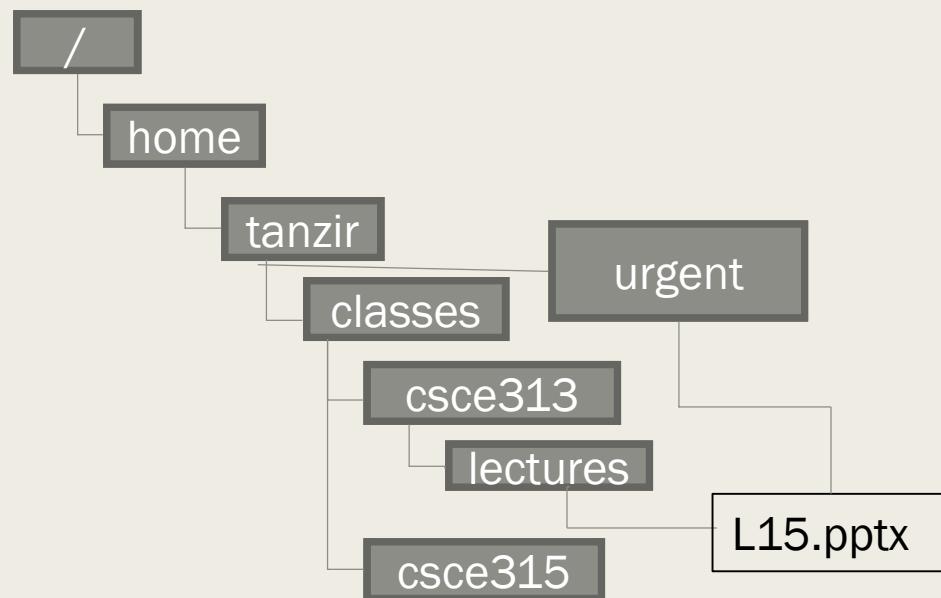


# File Systems Terms

- **path:** a string identifying a file (e.g.,  
`/home/tanzir/Work/hw1.cpp`)
- **root directory:** think of the directory as a tree whose root is the *root directory*
  - Often denoted by the “/”
- **absolute path:** a path starting with “/”
- **relative path:** a path relative to the **current working directory**. Does not start with a “/”

# File System “Tree”

- **hard link:** The mapping between the name and the underlying file
  - *There can be multiple hard links to the same file (e.g., short cuts)*
  - *Means that the directory tree is **not always a tree***



# UNIX Directory API: Current Directory

```
#include <unistd.h>

char * getcwd(char * buf, size_t size);
/* get current working directory */
```

Example:

```
void main(void) {
    char my cwd[PATH_MAX];

    if (getcwd(my cwd, PATH_MAX) == NULL) {
        perror ("Failed to get current working directory");
        return 1;
    }
    printf("Current working directory: %s\n", my cwd);
    return 0;
}
```

# UNIX Directory API – Open, Read, Close

- Read is **stateful** with a **cursor**
  - *Reading the same directory again gives back the next file in the directory*

```
#include <dirent.h>
int main(int argc, char * argv[]) {
    struct dirent * direntp;
    DIR * dirp = opendir(argv[1]);
    while ((direntp = readdir(dirp)) != NULL)
        printf("%s\n", direntp->d_name);

    closedir(dirp);
    return 0;
}
```

# UNIX Directory API – Traversal

- Read is stateful with a cursor
  - *Reading the same directory again gives back the next file in the directory*
  - *You can even do “seek” to the beginning using rewinddir()*

```
#include <dirent.h>

DIR* opendir(const char * dirname);
/* returns pointer to directory object */
struct dirent * readdir(DIR * dirp);
/* read successive entries in directory ‘dirp’ */
int closedir(DIR *dirp);
/* close directory stream */
void rewinddir(DIR * dirp);
/* reposition pointer to beginning of directory */
```

# File System Organization

- First, each sector/block is numbered from 0, 1, ....
  - *The larger the disk, the more the sectors*
- Then, the file system contains the following components:

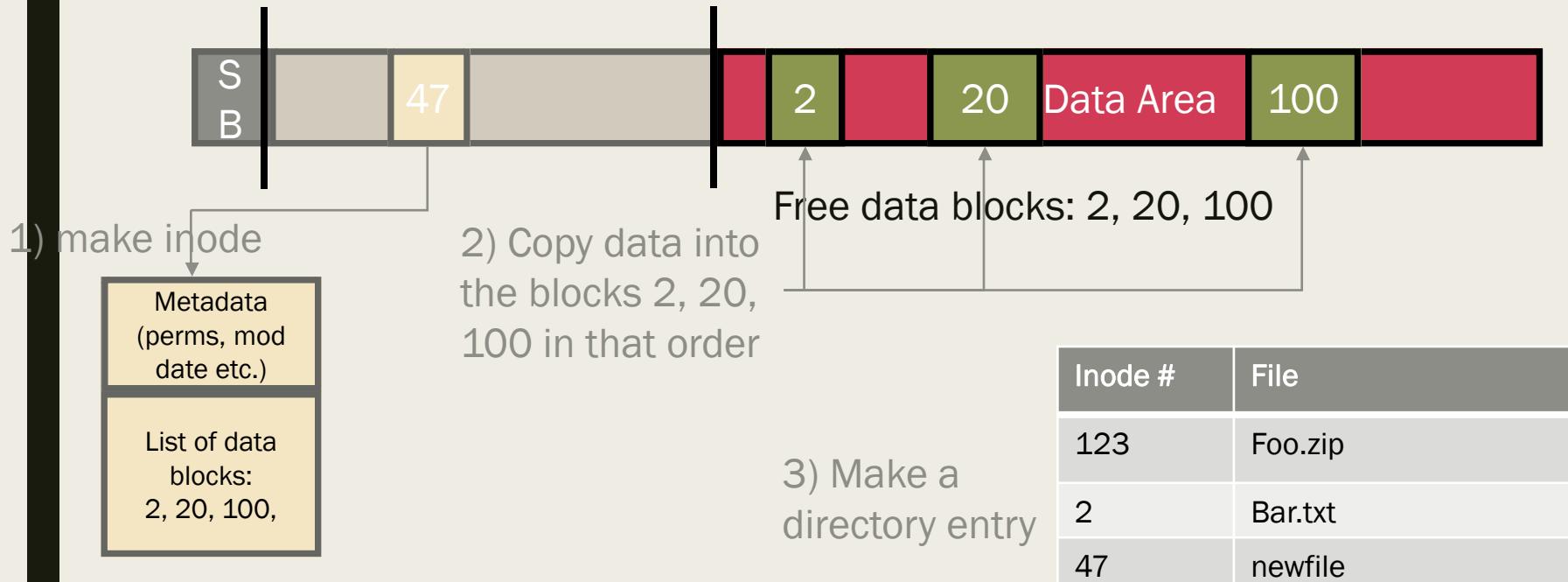
Component	Purpose
Superblock	Contains metadata about the file system. Size is OS dependent
Inode table	An array of inode structs, where each struct contains info of a file (e.g., size, owner id, last modification). Each inode has a number, which is the index into the inode table
Data Area	Contains file content. Each file can be $\geq 1$ block



# Creating a New File

Let us create a file called “**newfile**” that is 12KB in size, and a disk block is 4KB.

First, we need a free inode to put the file metadata, then find 3 free disk blocks to put the actual data



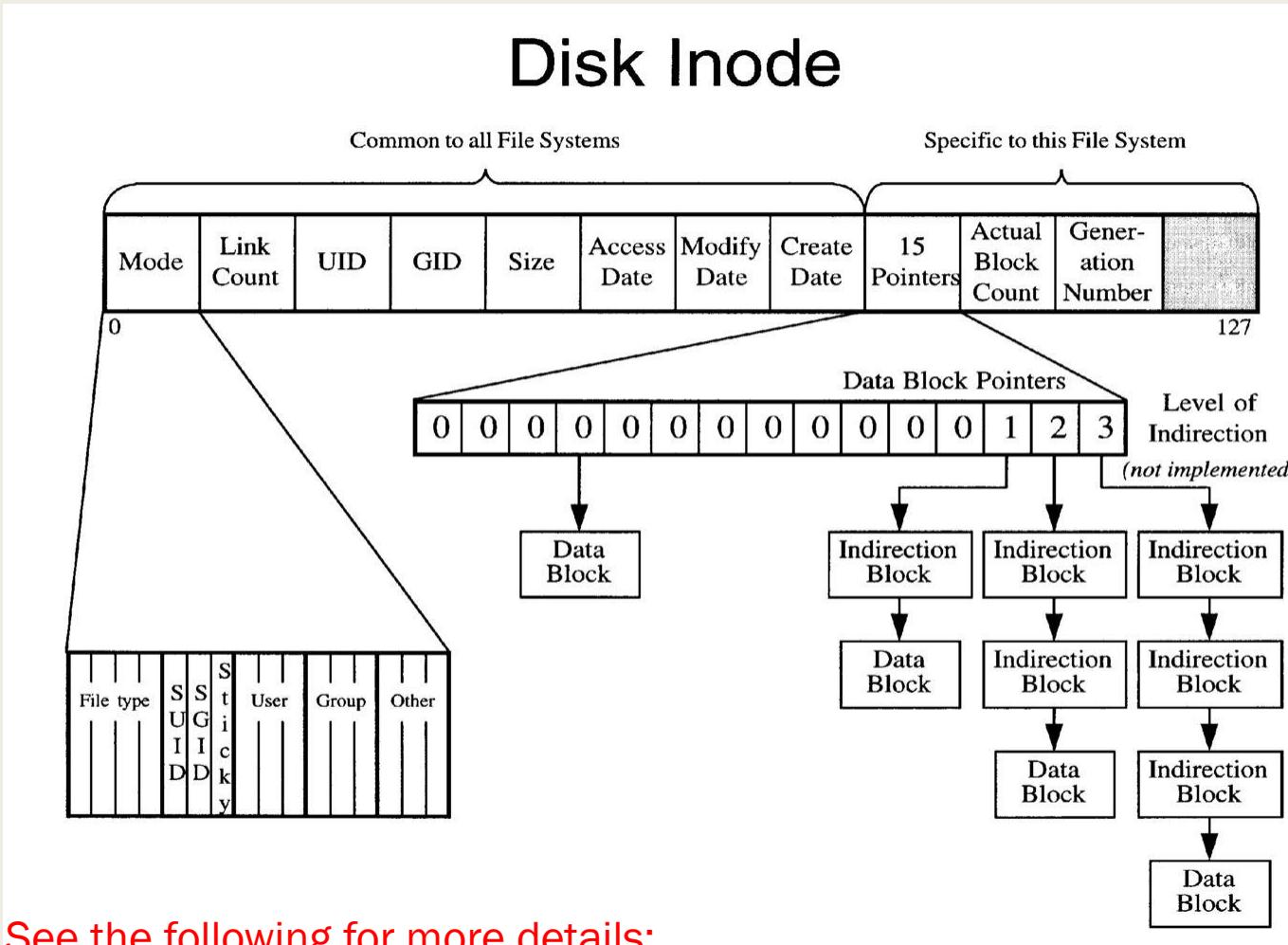
# Steps in Creating a File

- 1) Store Properties: Kernel looks for a free inode and stores the metadata (e.g., permissions, size, creation date) in that
- 2) Store Data and Record Allocations: Kernel then looks for free disk blocks (and enough of those) and copies content there. Kernel updates the inode with which blocks contain data for that file
- 3) Add file name to directory: Kernel stored the (inode#, filename) pair in the directory entry

# Reading a File

- 1) Search the directory for the file name and extract its inode
- 2) Locate and read the inode, find the data block number from there
- 3) Read each data block in sequence and output that
  
- This is how out “\$cat newfile” command will work
  - *Output will go to standard output*

# What goes inside a inode??



See the following for more details:

<http://man7.org/linux/man-pages/man7/inode.7.html>

# Inode's Features: Protection

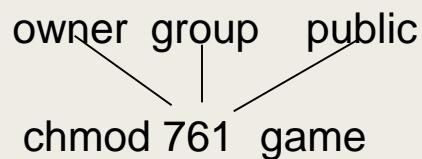
- File owner/creator should be able to control:
  - *what can be done*
  - *by whom*
- Types of access
  - *Read*
  - *Write*
  - *Execute*
  - *Append*
  - *Delete*
  - *List*

# Access Lists and Groups

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

		RWX
a) owner access	7	$\Rightarrow$ 1 1 1 RWX
b) group access	6	$\Rightarrow$ 1 1 0 RWX
c) public access	1	$\Rightarrow$ 0 0 1

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



Attach a group to a file: `chgrp G game`

NITIN AGRAWAL  
University of Wisconsin, Madison  
and  
WILLIAM J. BOLOSKY, JOHN R. DOUCEUR, and JACOB R. LORCH  
Microsoft Research

# Characteristics of Files

## Observations:

- *Most files are small*
- *Most of the space is occupied by the rare big ones*

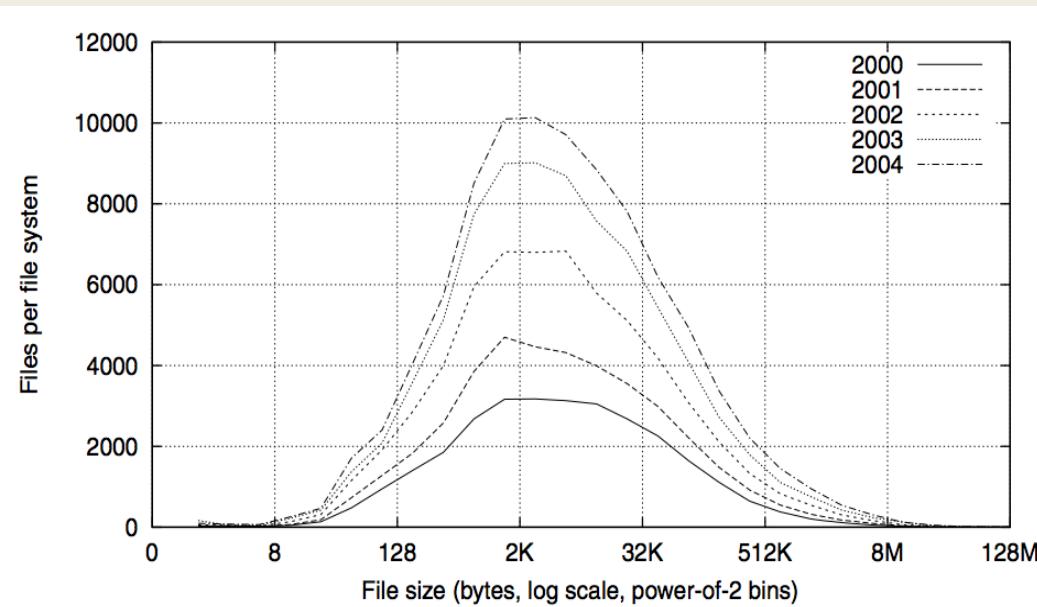
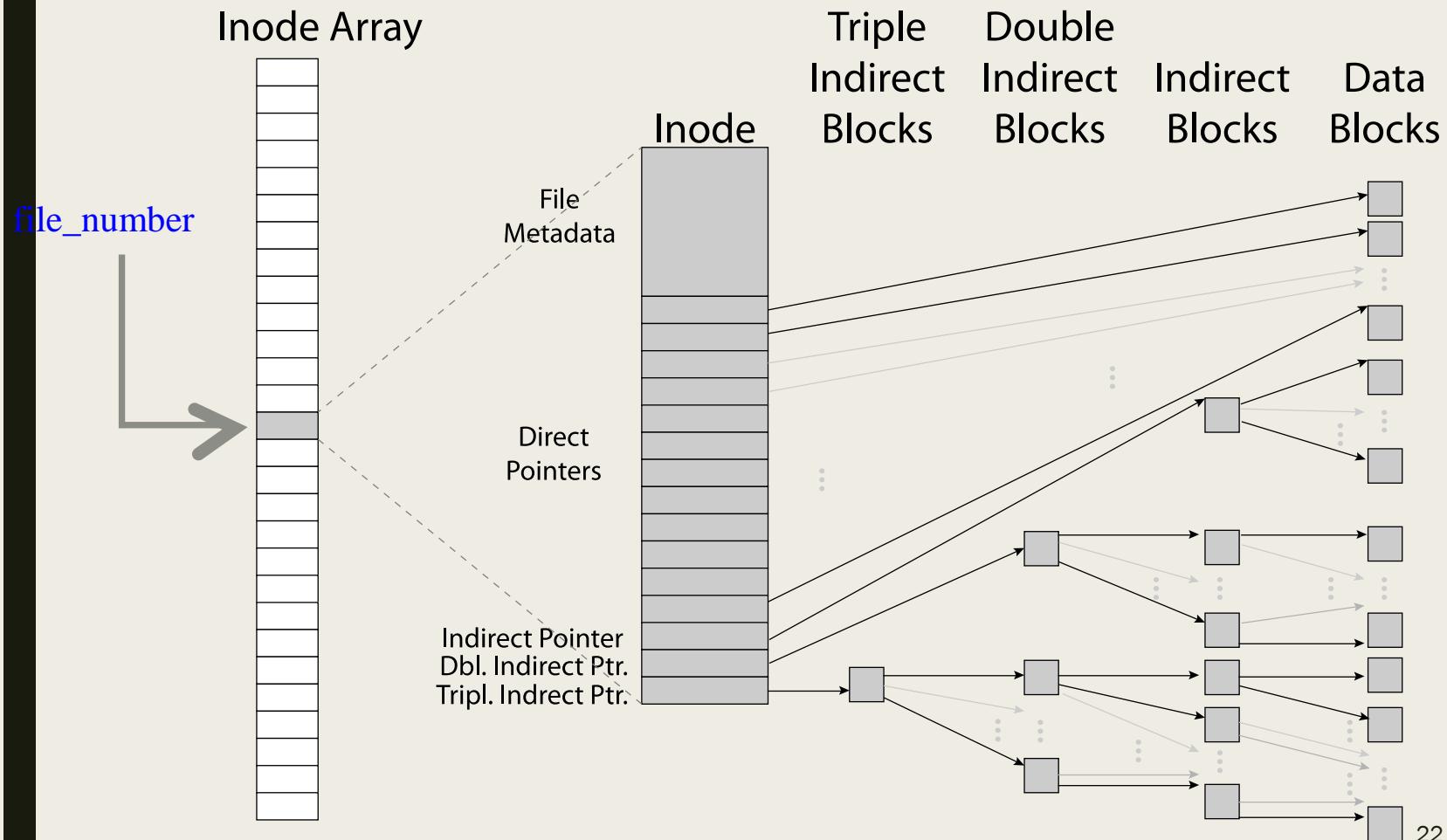


Fig. 2. Histograms of files by size.

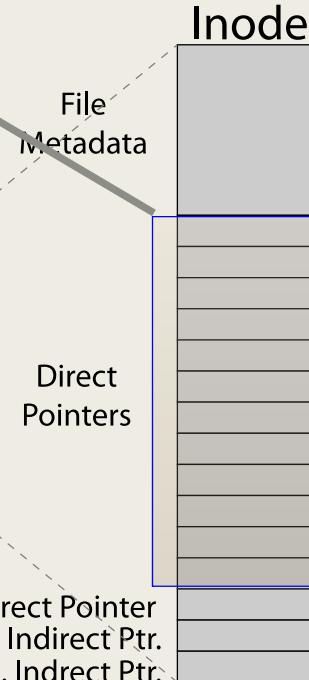
# Does this help deciding the inode structure?



# FFS: Data Storage

Small files: 12 pointers direct to data blocks

Inode Array



Direct pointers

With 4kB blocks, sufficient  
For files up to 48KB



Triple Indirect Blocks  
Double Indirect Blocks  
Indirect Blocks  
Data Blocks

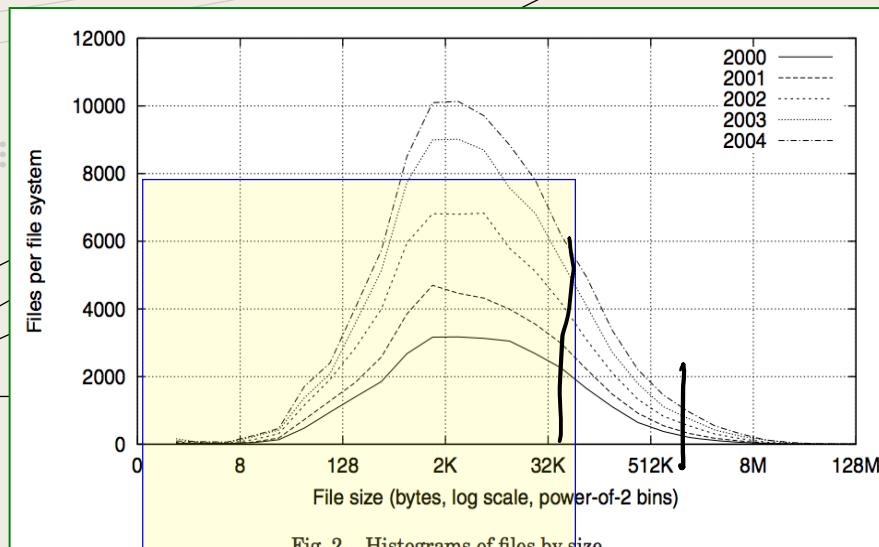
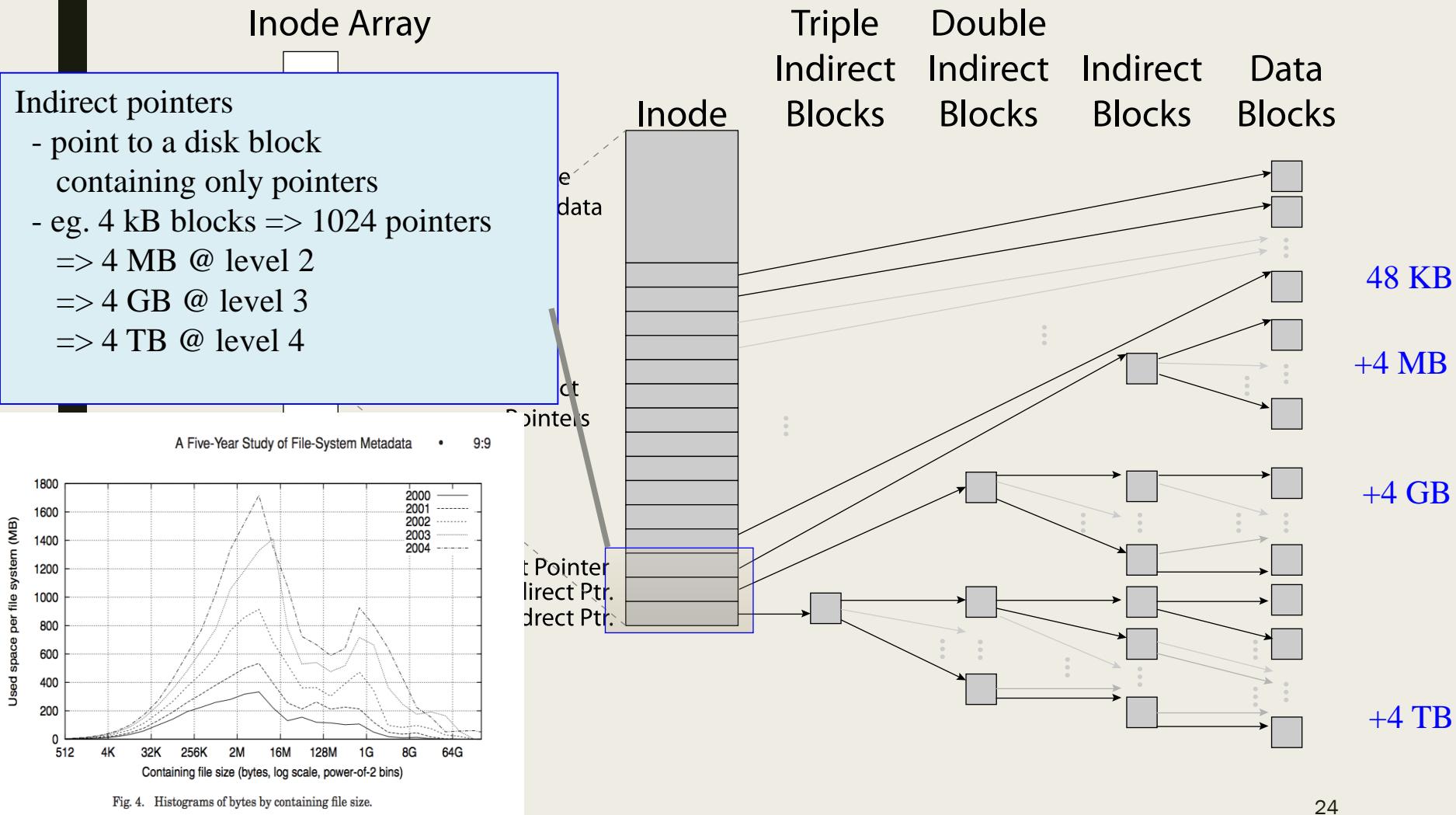


Fig. 2. Histograms of files by size.

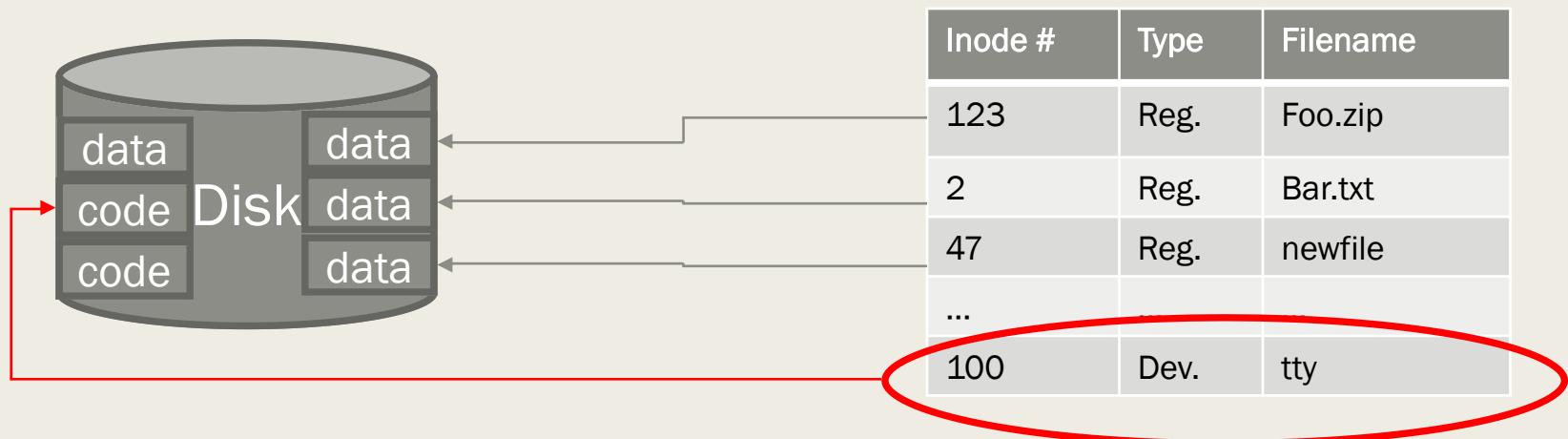
# FFS: Data Storage

- Large files: 1,2,3 level indirect pointers



# Device Files and inodes

- How to handle device files (e.g., terminal input, output, printer)?



Inode's for device files and regular files are different:

- They both point to disk blocks
- However, device files' inodes point to **device driver code** instead of regular data blocks

# Links

- Hard link
  - *Directory entry contains the **inode number***
  - *Creates another name (path) for the file*
  - *Each is “first class”*
- Soft link or Symbolic Link
  - *Directory entry contains the **name of the file***
  - *Map one name to another name*

# Hard Links

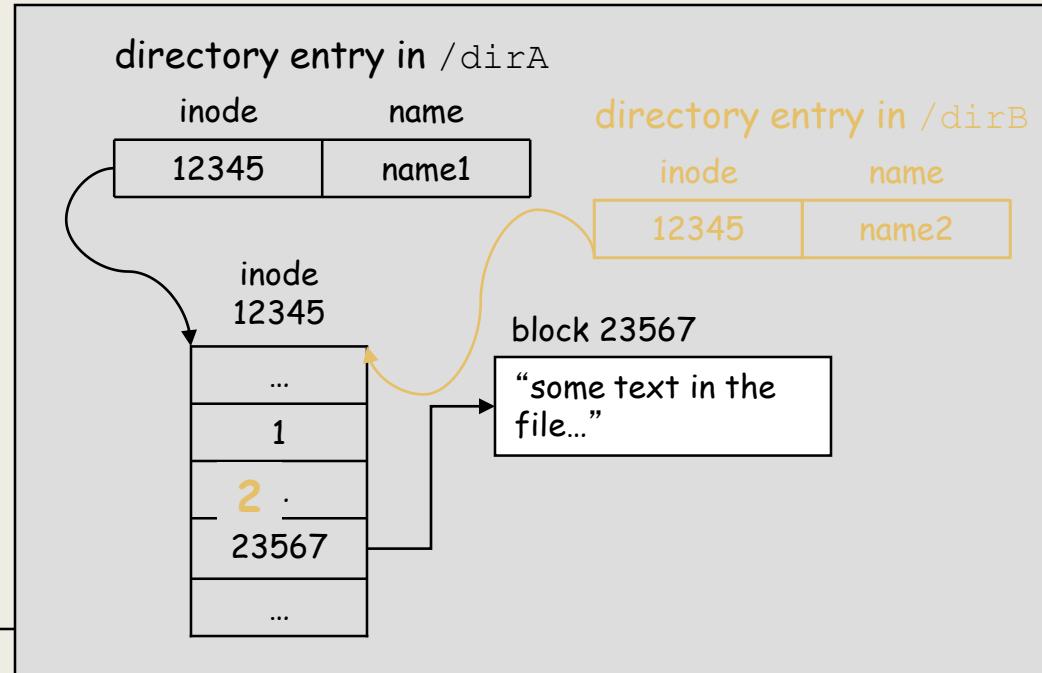
shell command

```
ln /dirA/name1 /dirB/name2
```

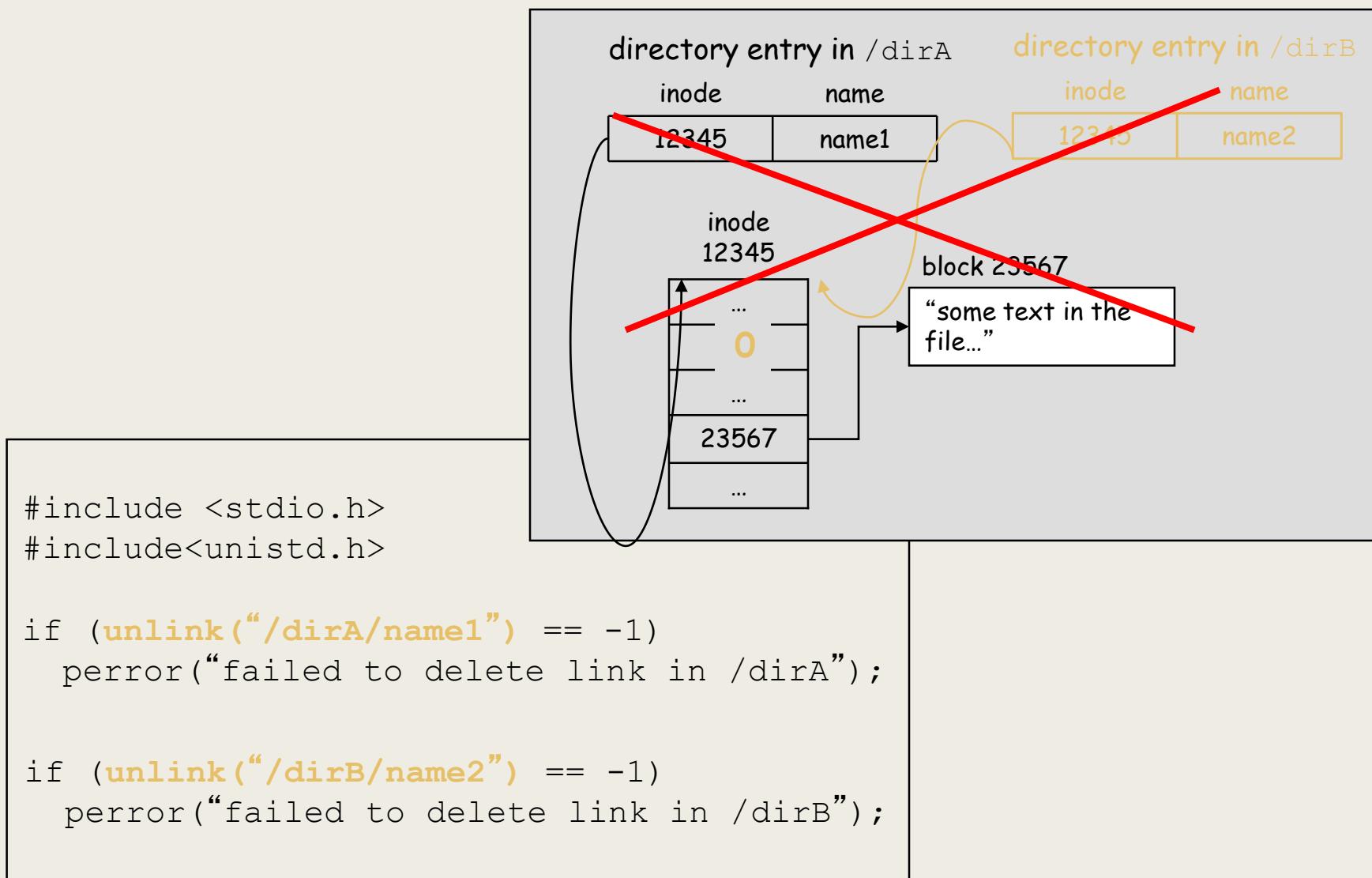
is typically implemented using the link system call:

```
#include <stdio.h>
#include<unistd.h>

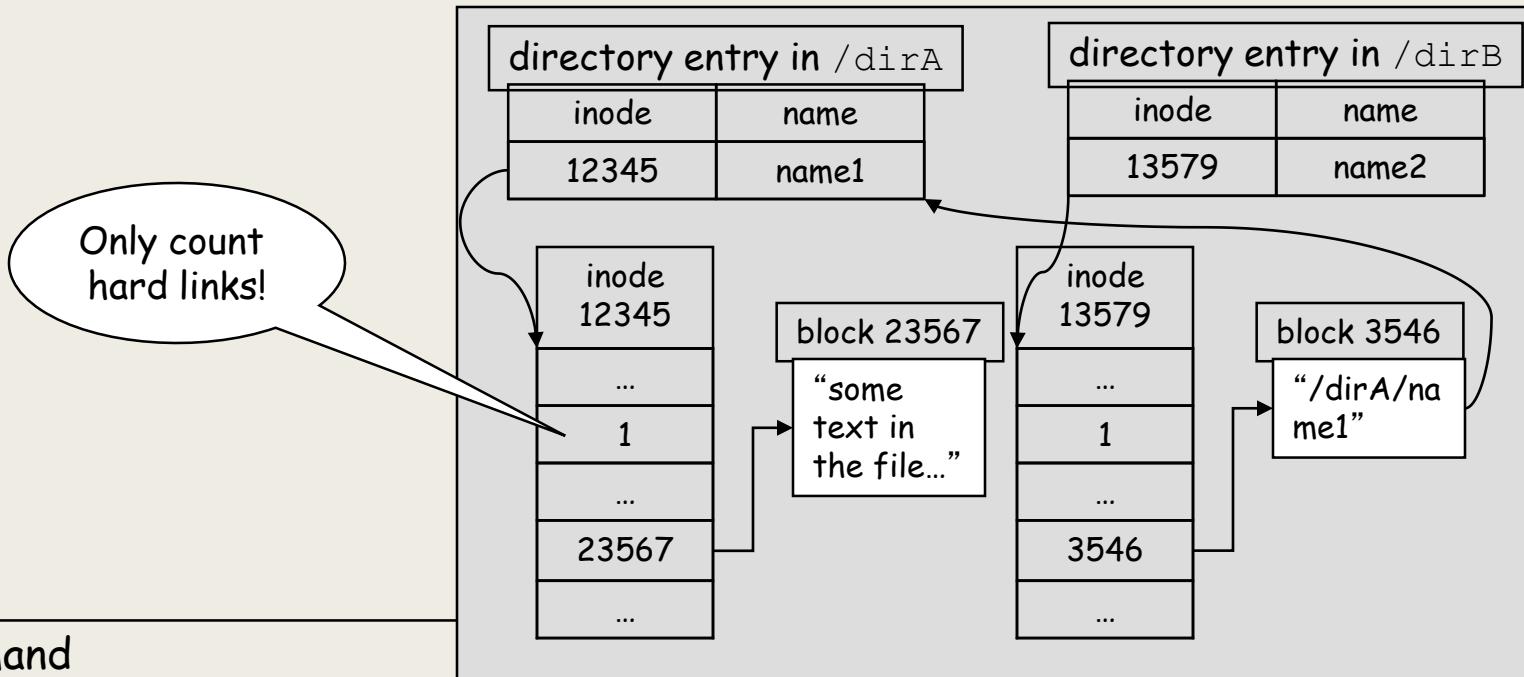
if (link("/dirA/name1", "/dirB/name2") == -1)
    perror("failed to make new link in /dirB");
```



# Hard Links: unlink



# Symbolic (Soft) Links



shell command

```
ln -s /dirA/name1 /dirB/name2
```

is typically implemented using the link system call:

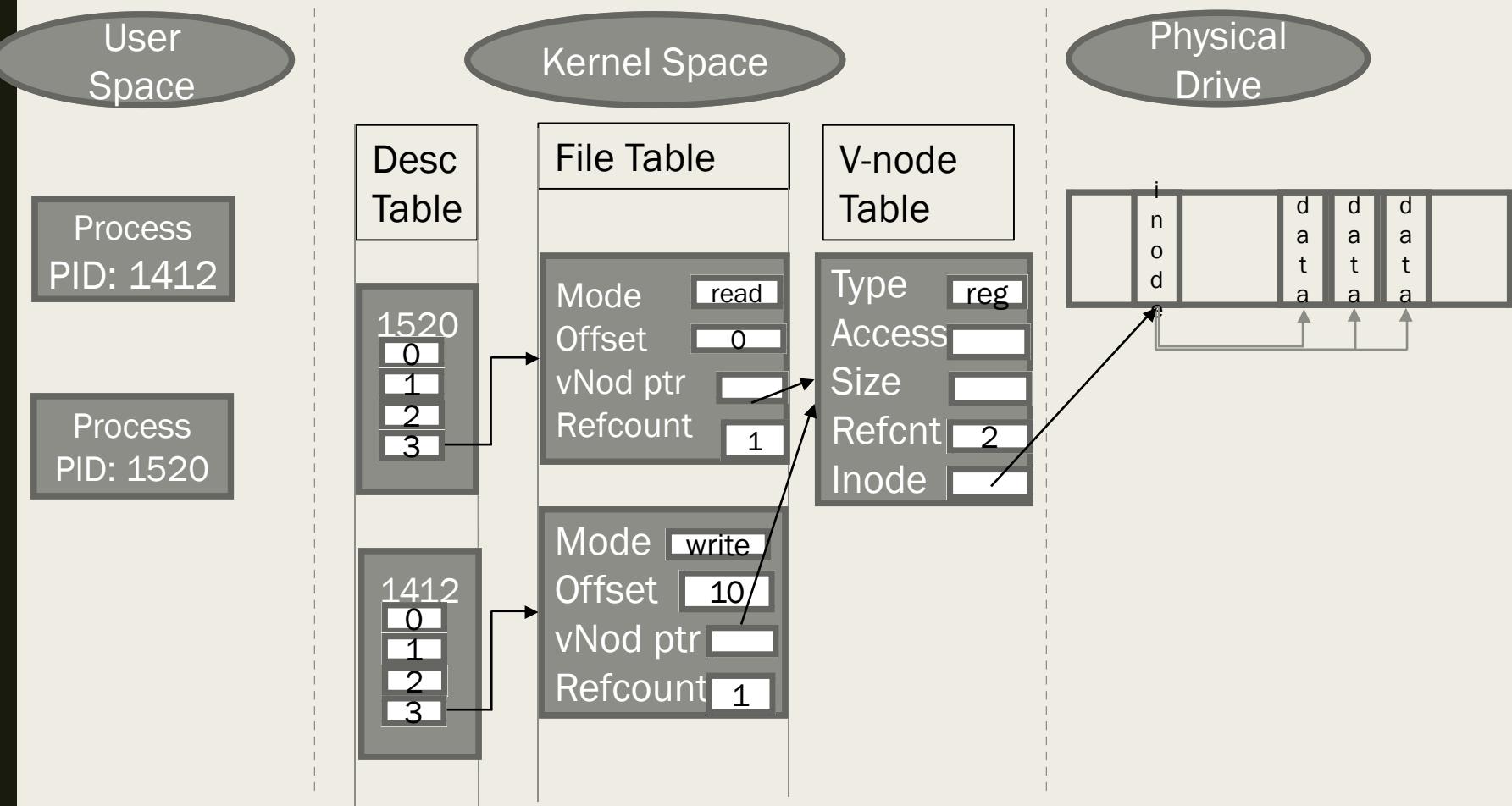
```
#include <stdio.h>
#include<unistd.h>

if (symlink("/dirA/name1", "/dirB/name2") == -1)
    perror("failed to create symbolic link in /dirB");
```

# Links: Example

```
[tanzir@compute PA7]$ ls -lrt
total 16
-rwxr-xr-x. 1 tanzir CSE_csfac 2251 Apr  7 03:21 semaphore.h
-rwxr-xr-x. 1 tanzir CSE_csfac  520 Apr  7 03:26 BoundedBuffer.h
-rwxr-xr-x. 1 tanzir CSE_csfac  765 Apr  7 03:28 BoundedBuffer.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac  533 Apr 19 20:20 makefile
-rwxr-xr-x. 1 tanzir CSE_csfac 1445 Apr 19 21:02 netreqchannel.h
-rwxr-xr-x. 1 tanzir CSE_csfac 3784 Apr 21 11:31 netreqchannel.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac 1012 Apr 21 11:46 dataserver.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac 7631 Apr 21 11:57 client.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac 1576 Apr 24 17:39 Untitled-1.cpp
[tanzir@compute PA7]$ ln semaphore.h ./sema.h
[tanzir@compute PA7]$ ln -s semaphore.h ./softsema.h
[tanzir@compute PA7]$ ls -lrt
total 18
-rwxr-xr-x. 2 tanzir CSE_csfac 2251 Apr  7 03:21 semaphore.h
-rwxr-xr-x. 2 tanzir CSE_csfac 2251 Apr  7 03:22 sema.h
-rwxr-xr-x. 1 tanzir CSE_csfac  520 Apr  7 03:26 BoundedBuffer.h
-rwxr-xr-x. 1 tanzir CSE_csfac  765 Apr  7 03:28 BoundedBuffer.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac  533 Apr 19 20:20 makefile
-rwxr-xr-x. 1 tanzir CSE_csfac 1445 Apr 19 21:02 netreqchannel.h
-rwxr-xr-x. 1 tanzir CSE_csfac 3784 Apr 21 11:31 netreqchannel.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac 1012 Apr 21 11:46 dataserver.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac 7631 Apr 21 11:57 client.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac 1576 Apr 24 17:39 Untitled-1.cpp
11wxrwxrwx. 1 tanzir games           11 Apr 26 19:34 softsema.h -> semaphore.h
```

# Files: Big Picture



# SIGNALS

Tanzir Ahmed  
CSCE 313 Spring 2020

# Background: User-Mode Exceptional Flow

- So far exceptional control flow features have been usable only by the operating system
  - *Exceptions:*
    - Synchronous: faults, traps, aborts
    - Asynchronous: interrupts
- All exception handlers run in **protected (Kernel) mode**
- Would like similar capabilities for **user mode code**
  - *Inter-Process communication to facilitate ‘exceptional control flow’ is subject of today’s discussion*
  - *We will discuss them through “Signals” mechanism*
    - .....but also in a broader context beyond just IPC

# What is a Signal?

- A Signal is a **one-word** message
  - *A Green Light is a signal, A Referee's whistle is a signal*
  - *These items and events do not contain messages, they are messages!*
- Each Signal has a numerical code
- So when we press CTRL-C key we ask the Kernel to send the interrupt signal to the currently running process

# List of Signals

Signal	Value	Action	Comment
<b>SIGHUP</b>	1	Term	Hangup detected on controlling terminal or death of controlling process
<b>SIGINT</b>	2	Term	Interrupt from keyboard
<b>SIGQUIT</b>	3	Core	Quit from keyboard
<b>SIGILL</b>	4	Core	Illegal Instruction
<b>SIGABRT</b>	6	Core	Abort signal from <code>abort(3)</code>
<b>SIGFPE</b>	8	Core	Floating-point exception
<b>SIGKILL</b>	9	Term	Kill signal
<b>SIGSEGV</b>	11	Core	Invalid memory reference
<b>SIGPIPE</b>	13	Term	Broken pipe: write to pipe with no readers; see <code>pipe(7)</code>
<b>SIGALRM</b>	14	Term	Timer signal from <code>alarm(2)</code>
<b>SIGTERM</b>	15	Term	Termination signal
<b>SIGUSR1</b>	30,10,16	Term	User-defined signal 1
<b>SIGUSR2</b>	31,12,17	Term	User-defined signal 2
<b>SIGCHLD</b>	20,17,18	Ign	Child stopped or terminated
<b>SIGCONT</b>	19,18,25	Cont	Continue if stopped
<b>SIGSTOP</b>	17,19,23	Stop	Stop process
<b>SIGTSTP</b>	18,20,24	Stop	Stop typed at terminal
<b>SIGTTIN</b>	21,21,26	Stop	Terminal input for background process
<b>SIGTTOU</b>	22,22,27	Stop	Terminal output for background process

Terminate and also dump core  
(for further investigation)

Will be ignored by default, unless you override

Process scheduler??

Source of table: <http://man7.org/linux/man-pages/man7/signal.7.html>

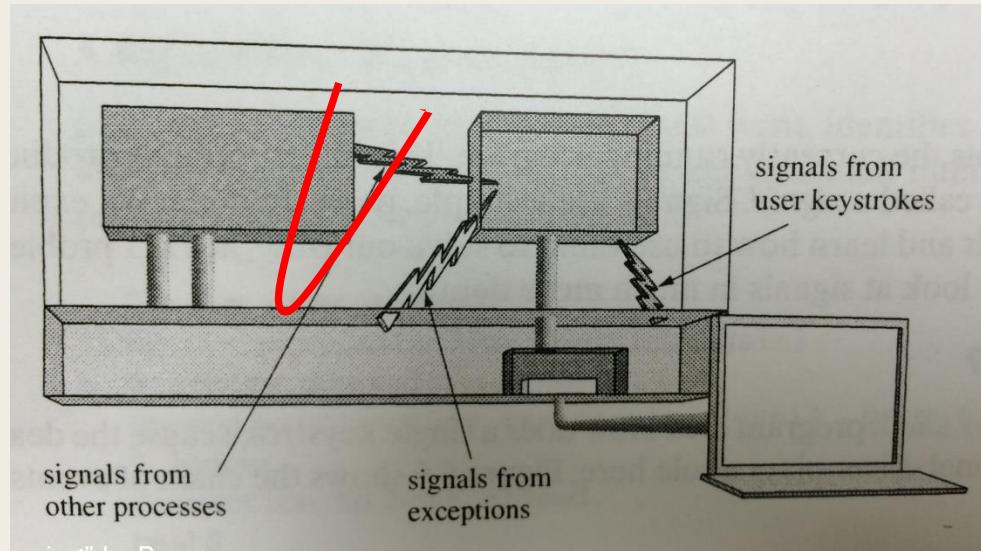
Multiple values mean that they are different based on architecture

# Signals – Some Points

- The “Action” in the previous page in many cases indicate the “Default Action”
  - You are **allowed to “override”** the action for some signals (e.g., **SIGINT**, **SIGUSRx**, **SIGTERM**, **SIGSTP**), while for others (**SIGKILL**, **SIGILL**, **SIGFPE**, **SIGSEGV**, **SIGSTOP**) you **cannot override**
  - *This is also called “signal handling”, or “catching”*
- SIGCHLD is the only signal so far that is **Ignored** by default
  - *Most others will kill the process except **SIGCONT** and **SIGSTP***
- Does that mean you can kill any process by sending any signal??
  - NO. You can only signal processes that you created
- **SIGSTP** (suspends a process) and **SIGCONT** (continues/unsuspends a process) are very useful for “job control”
  - *How to control a process group tied by a shell session?*
  - *All processes that you run make a group of processes that need special controlling*
  - *You may suspend a process, run it in the background, move to foreground*
  - *What happens to standard input/output of a background process?*

# Where do Signals come from?

- Today we'll look at Signals in a broader context
  - *IPC is one of the contexts (one process sending to another)*
- Others are facilitated by Users and Kernel
  - *[Users] Signals generated by external Input devices*
  - *[Kernel] Exceptions*



Taken from: Chapter 6 of “Understanding Unix/Linux Programming” by Bruce Molay

# Where do Signals come from?

(USER) Terminal-generated signals: triggered when user presses certain key on terminal. (e.g. **^C**)

*kill (2) function*: Sends any signal to another process.

*kill (1) command*: The command-line interface to *kill (2)*

*raise (3)* : Sends a signal to itself

*A user can send signals to only his owned processes*

(Kernel) Exception-generated signals: CPU execution detects condition and notifies Kernel. (e.g. **SIGFPE** divide by 0, **SIGSEGV** invalid memory reference, **SIGILL** when an instruction with illegal opcode is found)

(PROCESSES) Software-condition generated signals: Triggered by software event (e.g. **SIGURG** by out-of-band data on network connection, **SIGPIPE** by broken pipe, **SIGALRM** by timer)

# Generating Signals: kill(2) and raise(3)

```
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
    /* send signal ‘sig’ to process ‘pid’ */
```

```
/* example: send signal SIGUSR1 to process 1234 */
if (kill(1234, SIGUSR1) == -1)
    perror("Failed to send SIGUSR1 signal");

/* example: kill parent process */
if (kill(getppid(), SIGTERM) == -1)
    perror("Failed to kill parent");
```

```
#include <signal.h>
```

```
int raise(int sig);
    /* Sends signal ‘sig’ to itself.
        Part of ANSI C library! */
```

Raise sends a signal to the executing process  
Kill sends a signal to the specified process

# Signals and the Kernel

- Many of the signals are **generated by** the Kernel in response to events and exceptions received
  - *SIGFPE – FP exception*
  - *SIGILL – Illegal instruction*
  - *SIGSEGV – Segment Violation*
- All others are **routed through** the Kernel, if not originating from the Kernel itself
  - *E.g. SIGHUP (terminal hang), SIGINT (CTRL-C keyboard), SIGTSTP (CTRL-Z), SIGKILL (KILL)*

# What can a Process do about a Signal?

Tell the Kernel what to do with a signal:

1. **Accept Default action.** All signals have a default action signal (SIGINT, SIG\_DFL)
2. **Ignore the signal.** Works for most signals  
signal (SIGINT, SIG\_IGN)  
*cannot ignore SIGKILL and SIGSTOP; also unwise to ignore hardware exception signals*
3. **Catch the signal (call a function).** Tell the Kernel to invoke a given function (signal handler) whenever signal occurs.  
signal (SIGINT, foo)

# Simple Signal Handling: Example

```
linux2.cse.tamu.edu - PuTTY
```

```
:: more sigdemo.c
/* sigdemo.c - shows how a signal handler works
 * credit: Bruce Molay
 */

#include <stdio.h>
#include <signal.h>

main()
{
    void f(int); /* declare function */
    int i;

    signal(SIGINT, f);
    for (i=0; i<5; i++) {
        printf("hello\n");
        sleep(1);
    }
}

void f(int signum)
{
    printf("OUCH!\n");
}
```

The diagram illustrates the control flow between the `main()` function and its signal handler `f()`. It shows the normal flow of control from `main()` to `f()` via a signal, and the return flow from `f()` back to `main()`.

The `main()` function contains a loop that prints "hello\n" five times and sleeps for one second between each print. It also sets up a signal handler for SIGINT using the `signal` function.

The signal handler `f()` is a simple function that prints "OUCH!".

Annotations in the diagram:

- "normal flow of control" points to the initial entry into `main()`.
- "Arrival of SIGINT diverts control flow to the signal handler. Return from the handler resumes previous control flow." describes the signal delivery and return mechanism.
- "flow to signal handler and back" indicates the path from `main()` to `f()` and back to `main()`.

Taken from: Chapter 6 of “Understanding Unix/Linux Programming” by Bruce Molay

# Example – One Handler for Multiple Signals

```
static void sig_usr(int); /* one handler for two signals */

int main (void) {
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        perror("cannot catch signal SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        perror("cannot catch signal SIGUSR2");
    for(;;) pause();
}

static void sig_usr(int signo) { /*argument is signal number*/
    if (signo == SIGUSR1) printf("received SIGUSR1\n");
    else if (signo == SIGUSR2) printf("received SIGUSR2\n");
    else error_dump("received signal %d\n", signo);
    return;
}
```

# Signals: Terminology

- A signal is **generated** for a process when event that causes the signal occurs. (Hardware exception, software condition, etc.)
- A signal is **delivered** when action for a signal is taken.
- During the time between generation and delivery, signal is **pending**.
  - *That is also the **lifecycle** of a signal*
- A process has the option of **blocking** the delivery of a signal.
  - *Signal remains blocked until process either (a) unblocks the signal, or (b) changes the action to ignore the signal.*

**CSCE-313 Final Exam Fall 2015**

**Student Name** .....Aakash Tyagi.....

**Student ID** .....

**Instructions:**

1. This exam is worth 90 points and contains 9 questions.
2. Write your answers below each question in the space provided.
3. If you are unsure about your answer to a multiple choice or True/False statement, please write one sentence legibly explaining your assumption. Note that this is not required unless explicitly stated in the question itself.
4. Please take the first 5 minutes to read through the entire exam and strategize how you will proceed. The exam contains a mix of T/F, multiple choice, and problem solving questions. It is critical that you are able to assess their relative complexity and time-manage the completion of your exam.
5. **We are Aggies. We don't lie, cheat or steal, or tolerate those who do! Sign below showing your commitment and support for our code of honor.**

**Signature** .....

**Question 1. [10 points, 1 point each] Match each term in the left column to the definition and /or description in the right column that fits best. Do this by filling in the void entries on the left:**



(G) Pthread

(A) Lives in Kernel space and entries contain inode pointer among other attributes

(J) Unix Pipe

(B) Waits till all id'd signals are caught

(H) FIFO

(C) Mechanism for IPC via message passing

(I) UDP

(D) Synchronization primitive

(F) Socket

(E) Integer number identifying a file connection

(C) Mailbox

(F) Represented by address and port number

(N) SIGWAIT

(G) Thread that corresponds to POSIX standard

(D) vnode table

(H) Named IPC structure that resides in Kernel

(M) Mutex

(I) Protocol for Network Layer in Internet Programming

(E) File descriptor

(J) Mechanism for inter-process via the use of file descriptors

**Question 2. [10 points, 1 point each] Circle TRUE (T) or FALSE (F) for statements noted below:**

- a) T  F Doubling the block size of a Unix file system will double the max file size
- b) T  F A refcnt of 2 in a vnode table implies a file of size 2 blocks
- c)  T F Blocking a signal means to delay receipt of the signal until later
- d)  T F CIA triad in Security refers to Confidentiality, Integrity, and Availability of Data
- e)  T F A signal mask is a set of signals a process is ignoring
- f) T  F One major difference between datagram (UDP) sockets and stream (TCP) sockets is stream sockets are not reliable
- g) T  F A program can create a named pipe by calling "pipe" system call
- h) T  F Shared memory is faster for communication than pipes because the kernel stores shared memory segments in the CPU cache
- i)  T F Datagram sockets would be a good choice for transmitting background music for elevators instead of automated teller machine transactions
- j) T  F A critical section is a portion of memory that cannot be shared amongst processes.

**Question 3a. [5 points]** Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally.)

```
#include <stdio.h>
#include <signal.h>

pid_t pid;
void handler1(int sig) {
    printf("zip");
    fflush(stdout); /* Flushes the printed string to stdout */
    kill(pid, SIGUSR1);
}
void handler2(int sig) {
    printf("zap");
    exit(0);
}
main() {
    signal(SIGUSR1, handler1);
    if ((pid = fork()) == 0) {
        signal(SIGUSR1, handler2);
        kill(getppid(), SIGUSR1);
        while(1) {};
    }
    else {
        pid_t p; int status;
        if ((p = wait(&status)) > 0) {
            printf("zoom");
        }
    }
}
```

What is the output string that this program prints? EXPLAIN YOUR ANSWER. Correct answers WITHOUT explanation will only receive partial credit.

**zipzapzoom**

**Question 3b. [5 points, 2.5 points each]** Consider the following C program. (For space reasons, we are not checking error return codes, so assume that all functions return normally).

```
#include <stdio.h>
#include <signal.h>

int counter = 0;
void handler(int sig)
{
    counter++;
}

int main()
{
    int i;
    signal(SIGCHLD, handler);
    for (i = 0; i < 5; i++){
        if (fork() == 0){
            exit(0);
        }
    }
    /* wait for all children to die */
    while (wait(NULL) != -1);
    printf("counter = %d\n", counter);
    return 0;
}
```

**EXPLAIN YOUR ANSWERS. Correct answers WITHOUT explanation will receive partial credit.**

Note: When a child process stops or terminates, SIGCHLD is sent to the parent process. The default response to the signal is to ignore it. The signal can be caught and the exit status from the child process can be obtained by immediately calling wait.

**1. Does the program output the same value of counter every time we run it? Circle: Yes      No.**

**2. If the answer to the above is a Yes, indicate the value of the counter variable. Otherwise, list all possible values of the counter variable.**

Answer: counter = \_\_\_\_\_ **4, 5** \_\_\_\_\_

**Question 4. [10 points, 1 point each] Circle the best answer for each of the 10 multiple choice questions below. If in doubt about your answer, use the space below to state your assumption in 1 sentence.**

- A Unix directory contains
  - a. Names and inode number of files
  - b. Names and sizes of files
- Unix uses indirect blocks to store
  - a. symbolic links
  - b. lists of block numbers of data blocks
- The size of an inode table determines
  - a. the maximum number of files in the file system
  - b. the maximum size of a file in the file system
- A web server sends output of programs to the client by
  - a. opening a new socket for the program
  - b. using dup2 to redirect standard output
- One thing a server does that a client does not do is
  - a. a server loops
  - b. a server waits for a connection
- A server can handle several requests at once by
  - a. using fork to create a new process for each request
  - b. using socket to create a new socket for each request
- To prevent race conditions, threads should use
  - a. signals
  - b. mutex objects
- The input redirection symbol "<" is processed by
  - a. the program that reads the data
  - b. the shell
- The thread that accepts calls in a web server does not close the socket
  - a. because that would close it for the worker thread, too
  - b. because it is closed automatically when the worker thread finishes
- The shell notation "prog1 | prog2" is a request to
  - a. send all output from prog1 to the standard input of prog2
  - b. send the standard output of prog1 to the standard input of prog2

**Question 5a [6 points]** Consider a UNIX filesystem with the following components: Disk blocks are 4K Bytes. All pointers are 4 Bytes long. An inode has 12 direct block pointers, one indirect block pointer and one double-indirect block pointer. The total inode size is 256 Bytes. Both indirect and double indirect blocks holding pointers take up an entire block. How much disk space, including metadata and data blocks, is needed to store a 4 GB DVD image file? You can leave the answer in symbolic (e.g., 6MB + 3KB, or powers of 2) form— you must show your calculation for credit. For metadata, make reasonable assumptions.

Requires 232 bytes of actual data: 1048576 data blocks.

1024 block ptrs per indirect block.

$1,048,564 \text{ blocks} / 1024 = 1024$  indirect blocks needed.

1 double-indirect block needed.

256 bytes for inode.

Total: 4,299,165,952 bytes.

-2 for not including the actually data (4GB)

-1 for not including the inode (256B)

-1 for not including the doubly pointer block ( $1 * 2^{12}B = 4KB$ )

-1 for not including the singly (indirect) pointer blocks:

$(2^{10} * 2^{12}B = 4MB)$

-1 for each additional term.

-1 for not showing steps.

**Q5b [4 points]** Please draw the directory structure showing the result of executing the following

```
:: ls -lrt demodir/*
demodir/A:
total 1
-rw-r--r-- 1 tyagi CSE_csfac 0 Dec  9 07:20 a

demodir/B:
total 1
-rw-r--r-- 1 tyagi CSE_csfac 0 Dec  9 07:20 a
lwxrwxrwx 1 tyagi CSE_csfac 4 Dec  9 07:20 x -> ../x
```

stream of UNIX commands.

mkdir: creates a directory with the name supplied in the argument

rmdir: removes a directory with the name supplied in the argument

mv: moves a file or directory to new location (or name)

cd: change directory

Eg.1 User View of a FileSystem - Working example

```
% mkdir demodir
% cd demodir
% mkdir b oops
% mv b c
% rmdir oops
% cd c
% mkdir d1 d2
% cd ../..
% mkdir demodir/a
```

<space for drawing directory structure>

**Question 6a. [5 points]** A binary semaphore is initialized to TRUE. Then 5 P() operations are performed on it in a row followed by 8 V() operations. Now 5 more P() operations are performed on it. What is the number of processes waiting on this semaphore? Show your work.

*Sequence A [5 P() operations]: First one goes through and then 4 processes wait in the queue*

*Sequence B [8 V() operations]: All 4 waiting processes get released. Semaphore is set to TRUE*

*Sequence C [5 P() operations]: First one goes through and then 4 processes will be put on WAIT queue.*

**Question 6b. [5 points]** In class, we learnt about how we can protect a critical section by encasing it between *lock acquire* and *lock release* primitives. This allows only one process or thread to be able to enter the critical section at a time and prevents a possible race condition from occurring. One such mechanism for lock acquisition and release is through *disabling and enabling interrupts*. We also learnt that it is ‘responsible’ to enable interrupts *as soon as possible*. In the below pseudocode, there are three such opportunities for early enabling of interrupts. These are shown with markers A, B, and C. Markers point to the insertion of “enable interrupts”. Comment on each of them (a) whether the idea will be feasible or not, and (b) 1-2 line reasoning for your answer.

<pre> A   Enable Position → B   Enable Position → C   Enable Position →       Acquire() {           disable interrupts;           if (value == BUSY) {               put thread on wait queue;           } else {               value = BUSY;           }           enable interrupts;       }   </pre>	<pre>       Release() {           disable interrupts;           if (anyone on wait queue)               take thread off wait               queue           Put on the ready queue           } else {               value = FREE;           }           enable interrupts;       }   </pre>
---	--

Enabling interrupts at Marker A:

Not feasible, because *Release* can check the queue and not wake up thread until next lock acquire/release

Enabling interrupts at Marker B:

Not Feasible, because *Release* puts the thread on the ready queue, but the thread still thinks it needs to go to sleep; it misses wakeup and still holds lock (deadlock!)

Enabling interrupts at Marker C:

Not feasible, because once the thread goes to sleep, there is no way it could execute the instruction to enable the interrupts 😊

**Question 7. [10 points: 1+3+3+2+1] In the following pseudo code for a bounded buffer problem in a producer-consumer form for a beverage dispenser, please answer the following ensuing questions:**

```

Semaphore fullSlots = 0; // Initially, no coke
Semaphore emptySlots = bufSize;
Semaphore mutex = 1; // No one using machine

Producer(item) {
    emptySlots.P(); // Wait until space
    mutex.P(); // Wait until machine free
    Enqueue(item);
    mutex.V(); // Tell consumers there is
    fullSlots.V(); // more coke
}
Consumer() {
    fullSlots.P(); // Check if there's a coke
    mutex.P(); // Wait until machine free
    item = Dequeue();
    mutex.V(); // tell producer need more
    emptySlots.V(); // return item;
}

```

*Assume: Enqueue adds an item to the buffer, Dequeue does the opposite.*

**7a. What should the semaphores “emptySlots” and “fullSlots” be initialized to?**

Fullslots = 0; emptyslots = bufsize

**7b. Describe the function of the above code above by annotating each line above with comments.**

See above

**7c. Is the order of P important? State YES, NO, or MAYBE along with REASON.**

Yes. Can cause deadlock because the producer could hold the mutex and then do a P on emptyslots which if it was full will cause the producer to wait while still holding the mutex.

**7d. Is the order of V important? State YES, NO, or MAYBE along with REASON.**

No or Maybe. Mostly scheduling efficiency. It is mostly performance because as soon as you release the mutex, you are signaling that they can utilize the buffer. For example, if the buffer only was to have one item, a producer mutex v will signal to the consumer there is drink but in fact the producer had yet to do the V operation on fullslots.

**7e. Will this code work for more than one producer and one consumer?**

Yes. Because the code remains identical no matter how many P and C.

**Question 8a [5 points]** Consider the following code:

```
int main(int argc, char* argv[]) {  
    char buf[] = "ab";  
    int r = open("file.txt", O_RDONLY);  
    int r1, r2, pid;  
    r1 = dup(r);  
    read(r, buf, 1);  
    if((pid=fork())==0) {  
        r1 = open("file.txt", O_RDONLY);  
    }  
    else{  
        waitpid(pid, NULL, 0);  
    }  
    read(r1, buf, 1);  
    printf("%s", buf);  
    return 0;  
}
```

Assume that the disk file file.txt contains the string of bytes 15213 . Also assume that all system calls succeed. What will be the output when this code is compiled and run? Explain your answer.

Reference: (a) **waitpid()** system call suspends execution of the calling process until a child specified by pid argument has changed state. (b) **dup(fd)** copies the given file descriptor fd into the lowest-numbered available file descriptor and then returns the new file descriptor.

**Answer: 1b5b**

**Question 8b [5 points]** Consider the following code

```
int main(int argc, char* argv[]) {  
    char buf[] = "abc";  
    int r = open("file.txt", O_RDWR);  
    int r1 = 0;  
    int r2 = open("file.txt", O_RDWR);  
    dup2(r, r1);  
    read(r, buf, 1);  
    read(r2, buf, 2);  
    write(r, buf, 3);  
    read(r2, buf, 1);  
    write(r1, buf, 1);  
    return 0;  
}
```

Assume that the disk file **file.txt** originally contains the string of bytes **12345**. Also assume that all system calls succeed. **What will file.txt contain when this code is compiled and run? Explain your answer.**

**Answer: 112c2**

**Question 9. [10 points] Circle the best answer for each of the 10 multiple choice questions below. If in doubt about your answer, use the space below to explain in less than 2 sentences.**

1. Shared memory is faster for communication than pipes because
  - a. the kernel stores shared memory segments in the CPU cache
  - b. data does not have to be copied from one process to another
2. A program deletes a shared memory segment by using
  - a. shmdt
  - b. shmctl
3. If two different processes write to the same pipe,
  - a. data from one writer may overwrite data from the other one
  - b. both sets of messages will get to the reader
4. If two different processes read from the same file,
  - a. both will read the same data
  - b. one will get some data, the other will get the rest
5. If two different processes read from the same pipe,
  - a. both will read the same data
  - b. one will get some data, the other will get the rest
6. A named pipe is removed with the system call
  - a. unlink
  - b. close
7. A client program makes a connection to a server program by calling
  - a. socket
  - b. connect
8. The listen system call is used to
  - a. receive data from a client
  - b. enable incoming connections to a socket
9. A pipe can be used to transfer data between
  - a. only two processes
  - b. any number of processes
10. When setting up a pipe to a child, the order of system calls is
  - a. the parent calls pipe then calls fork
  - b. the parent calls fork, then both processes call pipe

# Final Test

CSCE 313 Fall 2016  
(100 points)

[This exam has 6 question spanning 6 pages] Date:  
December 9, 2016

Student Name: .....

UIN:.....

I have adhered to Aggie Code of Honor

Signature:.....

**Question 1 [25 points] Circle True (T) or False (F) for the following statements**

- a) T F `read(fd,buf,x)` advances the file cursor by `x` bytes
- b) T F `close()` function is automatically called on the open file descriptors when a program exits
- c) T F Making a symbolic link increases the inode's reference/link count
- d) T F The function `listen()` on the server side attaches the process to a port
- e) T F The function `getaddrinfo()` is used on the client side to look up the server's IP address from its name
- f) T F `open()` function always creates a new entry in the File Table
- g) T F Locks with atomic `test&set()` will always require some busy-wait
- h) T F `cout` function flushes its buffer upon seeing a "\n"
- i) T F Shared memory IPC is faster because it does not actually copy data between address spaces
- j) T F No data is actually written to the disk when using FIFO IPC
- k) T F `open()` function is used to create a FIFO
- l) T F `msgget()` is used by both the sender and the receiver in message passing
- m) T F A shared memory segment is maintained by the process creating it
- n) T F In shared memory IPC, the same physical memory is mapped to the address spaces of more than 1 processes with the help of virtual memory
- o) T F A TCP connection is a connection between two IP addresses and port number tuples
- p) T F The purpose of standard I/O is both code portability and buffering facility
- q) T F inodes in a system are fixed in size and kept in a inode array
- r) T F A blocking send function will wait until the full data is delivered/buffered
- s) T F `sigprocmask()` is used by a process to block/unblock signals
- t) T F Processes maintain one bit vector for the blocked signals and another for the pending ones
- u) T F A Unix directory entry contains both the file name and the inode number
- v) T F A Superblock contains metadata about the file system
- w) T F A inode of a network socket file contains pointers to device driver code
- x) T F A hard link to the original file does not share the same inode
- y) T F A soft link keeps the new name in the directory entry and the original file name in the data block

**Question 2 [20 points]** A Linux file system has inodes that are each 128 bytes and each include 12 direct, 1 single indirect and 1 double indirect pointer. Now, a disk using this file system is formatted using 4KB data blocks and contains files of only 3 different sizes. The following table lists the number of files under each of these file sizes:

File Size	Number of files of this size
48KB	$2^{20}$ (= 1,048,576)
4MB + 48KB	$2^{10}$ (= 1,024)
4GB + 4MB + 48KB	2

Assuming each pointer is 4 bytes, compute the total **overhead** of this disk to store the above files. Here, the overhead is the amount of disk space required for keeping the **inodes** and the **pointers** to data blocks (i.e., non-data or pointer-only blocks in case of single and double indirect).

We have 3 types of files 1) 48KB, 2) 48KB + 4MB, and 3) 48KB + 4MB + 4GB. Let us do separate calculation for them and then do weighted sum on them.

Assumption: I am not counting inode's space as overhead (which you may or may not do.).

#### 1) 48KB

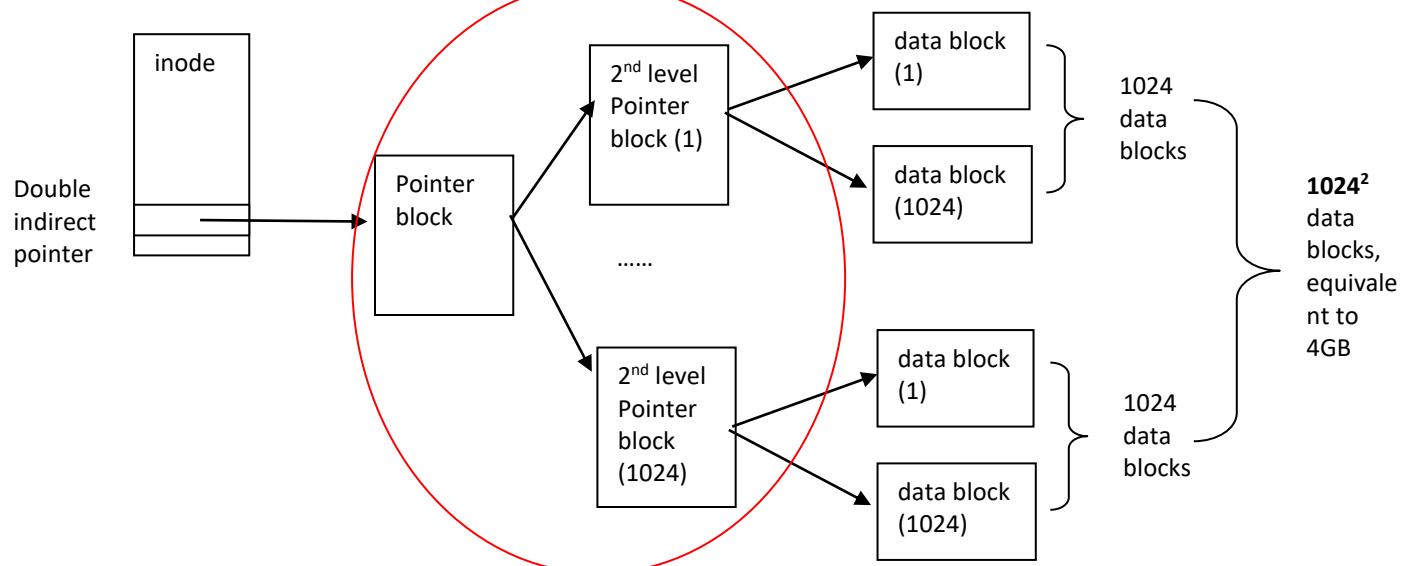
Only the 12 block pointers inside the inode are enough for this type of file ( $12 \times 4KB = 48KB$ ). So, excluding the inode, **Overhead = 0**.

#### 2) 48KB + 4MB

These will need the 12 in-inode pointers and 1024 pointer to data blocks ( $1024 \times 4KB = 4MB$ ). We can fit all these 1024 pointers in a single indirect block because a block of 4KB can keep 1024 (= 4MB/4) pointers. So, overhead in this case is = 1 block worth of pointers or just **4KB**

#### 3) 48KB + 4MB + 4GB

To keep a 4GB file, we need  $1024^2$  pointers (because  $4KB * 1024^2 = 4GB$ ). For that, we need 1 double indirect pointer that points to 1 pointer block, which points to 1024 pointer blocks, each of these eventually point to 1024 data blocks, forming a tree structure like the following:



So, the overhead here is all the pointer blocks. These blocks form a tree with a root node and 1024 leaf nodes (shown in red circle above). So, total there are  $1024+1 = 1025$  pointer blocks. Alternatively, the overhead is:

$1025 \times 4KB$  (for 4GB) + 4KB (for the rest 48KB+4MB) = **1026\*4KB**

Now, the total overhead for the entire file system is:

$0 * 2^{20} + 4KB * 210 + 1026*4KB * 2$

**Question 3 [20 points]:** For the given program below, do the following:

1. Draw the descriptor table and file table after executing lines 4, 7, 11
2. Write what is printed in the standard output accurately (cannot print unnecessary double quotes or newlines if they are not printed from the program).

Note that the file foobar.txt contains “foobar”. When you draw the tables, please include the reference count and file position fields in the file table. Also draw the arrows going from the descriptor table to the file table. Omit the v-node table.

```
1 int main () {
2     char c;
3     int fd = open ("foobar.txt", O_RDONLY);
4     dup2 (fd, 0);           // draw tables
5     if (!fork ()) {
6         cin >> c;
7         cout << c << endl; // draw tables
8     } else {
9         wait (0);
10        read (fd, &c, 1);
11        cout << c << endl; // draw tables
12    }
13    lseek (0, 0, SEEK_SET);
14}
```

(NOTE: This question is not part your final syllabus in Spring 2018, because it was covered in midterm)  
(P.S. I did not draw the pictures but you have to)

After line 4, STDIN is rewired to fd which is the file. But remember that “STDIN” and fd share the same File Table entry - so there is only one offset.

After line 6, 1 byte is read from the files which is “f” and it is printed in line 7.

After that (I say “after that” because there is a `wait(0)` in the parent process so we are making sure that the child process lines 6-7 ran first) in line 10, the next character “o” will be read and then printed in stdout. The file cursor is now at the 3<sup>rd</sup> character in the file.

Finally, in line 13, when you lseek() the STDIN, the file cursor now points to the first character "f" again, because through STDIN, you are actually modifying the File Table entry of the fd file.

**Question 4 [5 points]:** In the `Producer()` function of the `BoundedBuffer` implementation, can we reorder lines 3 and 4? In other words, can we do the `mutex.lock()` first and then `emptySlots.P()`? Describe why, or why not. [You will not get any points if you just write either "Yes" or "No". You have to provide reasoning].

```
1 void Producer(item)
2 {
3     emptySlots.P();
4     mutex.P();
5     Enqueue(item);
6     mutex.V();
7     fullSlots.V();
8 }
```

Already explained in Quiz 4 (Spring 2018)

**Question 5 [15 points]:** The signal SIGINT is generated when Ctrl+C is pressed from keyboard. Now, observe the following program and answer a), b) and c) below.

```
1 void exception_handler (int signo){
2     printf ("Interrupted\n");
3 }
4 int main () {
5     signal (SIGINT, exception_handler); //comment for c)
6     for (int i=0; i<5; i++){
7         printf ("Hello World\n");
8         sleep (2);
9     }
10 }
```

a) [5 pts] What is the output of the following program without any key press? How long will it run for?

It will run for 10 seconds. And it will print "Hello World\n" 5 times

b) [5 pts] What is the output if you press Ctrl + C once after 3 seconds of starting the program? How long will the program run now?

Will print the following and will run for the same amount of time (10 sec)

```
Hello World
Hello World
Interrupted
Hello World
Hello World
Hello World
```

c) [5 pts] Repeat a) and b) with line 5 commented out?

The output of a) will stay same and run for 10 secs.  
Output of b) will as follows:

```
Hello World  
Hello World  
^C
```

Then it will exit after 3 seconds from the start, because Ctrl+C kills the program.

**Question 6 [15 points]:** In the following, you are given the pseudocode for a Producer and a Consumer thread. They share a global variable “**Data d**” to exchange data, but are not synchronized yet. You have to rewrite the following code with proper synchronization primitives (e.g., mutex/semaphore) such that:

- There is no race condition
- Data is always produced first by the producer and then consumed by the consumer

[Hints: You can use 2 semaphore/mutex with proper initialization]

```
Data d;  
ProducerThread () {  
    while (true)  
        ProduceData (&d);  
}  
  
ConsumerThread () {  
    while (true)  
        ConsumeData (&d);  
}
```

---

Since there is just 1 producer and 1 consumer, we do not need a mutex which is generally used in a BoundedBuffer code. Also, we can use a bounded buffer of capacity=1.

```
Semaphore full = 0, empty = 1;  
Data d;  
ProducerThread () {  
    while (true){  
        empty.P();  
        ProduceData();  
        full.V();  
    }  
}  
ConsumerThread () {  
    while (true){  
        full.P();  
        d = ConsumeData();  
        empty.V();  
    }  
}
```

**CSCE-313 Final Exam Spring 2016****KEY****Student Name .....****Student ID .....****Instructions:**

1. This exam is worth 60 points and contains 6 questions with multiple parts.
2. Write your answers below each question in the space provided.
3. If you are unsure about your answer to a multiple choice or True/False statement, please write one sentence legibly explaining your assumption. **Note that this is not required unless explicitly stated in the question itself.**
4. Please take the first 5 minutes to read through the entire exam and strategize how you will proceed. The exam contains a mix of T/F, multiple choice, and problem solving questions. It is critical that you are able to assess their relative complexity and time-manage the completion of your exam.
5. **We are Aggies. We don't lie, cheat or steal, or tolerate those who do!**  
**Sign below showing your commitment and support for our code of honor.**

**Signature .....**

**Question 1a.** [5 points, 0.5 point each] Match each term in the left column to the definition and /or description in the right column that fits best. Do this by filling in the void entries on the left:

- |                      |   |
|----------------------|---|
| [E] Critical Section | A. has a unique inode                                     |
| [I] Unix Pipe        | B. CTRL-Z   |
| [H] SIGINT           | C. contains metadata, refcnt, and reference to inode      |
| [B] SIGSTP           | D. shares an inode with other file                        |
| [F] inode table      | E. piece of code that only one thread can execute at once |
| [A] Soft Link        | F. contains an array of inode structs                     |
| [J] File Descriptor  | G. Unix file  |
| [D] Hard Link        | H. keyboard interrupt                                     |
| [G] tty              | I. mechanism to communicate via file descriptors          |
| [C] vnode table      | J. integer number identifying a file connection           |

**Question 1b.** [5 points, 0.5 point each] Circle TRUE (T) or FALSE (F) for statements noted below:

- a)  T      F      A threat is a potential violation of security
- b)  T      F      The ref count of the inode of a directory should never be larger than 1
- c) T       F      The inode of a symbolic link contains the inode # of the target of the link
- d)  T      F      Compilers can optimize and reorder threaded code
- e)  T      F      An attack is any action that violates security
- f) T       F      A program can create a named pipe by calling "pipe" system call
- g) T       F      Datagram sockets would be a good choice for performing bank transactions
- h)  T      F      Doubling the block size of a UNIX file system with direct pointers will double max file size
- i) T       F      When analyzing threats, the cause, result, and intention are equally important
- j)  T      F      When we delete a hard linked file, the inode attached to that file remains intact

\* F is OK if the student says that an administrator has overruled the policy that prevents hard links for directory. If an explanation does not accompany the choice of "F" then point must be deducted.

\*\* The inode never contains the inode # of the target. It always points to blocks.

**Question 2a.** [5 points, 2+3 points respectively] We discussed the function `dup2(int filedes, int filedes2)` that duplicates an existing file descriptor. A variation of this function is `dup(int fildes)`, which copies the given file descriptor into the lowest-numbered available file descriptor and then returns the new file descriptor. Assume a process that executes the following three function calls:

```
fd1 = open(pathname, oflags);
fd2 = dup(fd1);
fd3 = open(pathname, oflags);
```

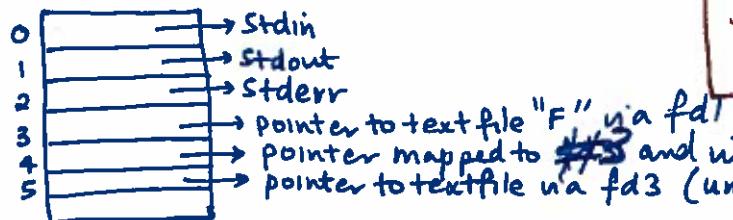
**2a.1** Draw a picture showing the connections of the file descriptor table to the process table.

**2a.2** Assume that the file identified by pathname is a text file with the following content: "abcdefghijklmнопqrstu". What would be the output of the following code snippet if it were to execute after the sequence of function calls above? (Assume: the function `read_char()` returns the next character from the given file).

```
char c1 = read_char(fd1);
cout << c1;
char c2 = read_char(fd2);
cout << c2;
char c3 = read_char(fd3);
cout << c3;
```

Write the final result of executing the above code snippet and explain briefly how you arrived at the answer.

**2a.1 (2points)**



Rubric :

+1 for showing only a partial pi  
+2 for showing the 3 additional file descriptors connected to file table

**2a.2 (3points)**

fd1 access will print out "a"  
fd2 access will print out "b" since fd2 shares file connection of fd1 & will be offset.  
fd3 access will print out "a" since it will start from beginning of the file

Output will be aba

Rubric

+1.5 for correct answer without explanation

+3 for correct answer with explanation (1 point for each correct step)

Question 2b. [5 points] Consider the following code:

```

int main(int argc, char* argv[]) {
    char buf[3] = "aha";
    int r = open("file.txt", O_RDWR);
    int r1 = 0;
    int r2 = open("file.txt", O_RDWR);
    dup2(r, r1);
    read(r, buf, 1);
    read(r2, buf, 2);
    write(r, buf, 3);
    read(r2, buf, 1);
    write(r1, buf, 1);
    return 0;
}

```

Main

+2.5 points for correct answer  
without explanation  
 +5.0 points for correct answer  
 with moderate explanation  
 → +1.0 for every correct step  
 (total 5 steps)

Assume that the disk file `file.txt` originally contains the string of bytes `csce313`. Also assume that all system calls succeed. What will `file.txt` contain when this code is compiled and run? Explain your answer.

`r`, `r1`, and `r2` are the file descriptors and are assigned the respective entries in the file descriptor table.

`r` and `r2` are given unique links to the file table and attached to the file "`file.txt`" with RD & WR perms.  
`r1` gets to share the link with `r` through call `dup2`

1. read will bring the character `c` into `buf` which will now look `cha`. The fd position is incremented to #2
2. read on `r2` will bring the characters `c & s` into `buf` which will now look as: `csa`. The fd position is incremented to #2
3. write will take 3 characters from `buf` and put them in the file after at position 2 and onwards. So `file.txt` will contain `ccsa313`. The position is advanced to #5
4. read<sup>on `r2`</sup> will change byte character `s` and put it in `buf` which will now look as `gsa`
5. write will take 1 character from `buf` and put it in file to read `ccsa513`

## Rubric:

+1 points for correct answers

+3 points for correct explanation accompanying answers

**Question 3a. [7 points]** You are to modify an ancient file system implementation that is structured as follows. Each inode contains 13 block numbers (pointers) of 4B (bytes) each; the first 10 block numbers point to the first 10 blocks of the file, and the remaining 3 are used for the rest of the file. The 11<sup>th</sup> block number in the inode points to an indirect block containing 128 block numbers, the 12<sup>th</sup> block number in the inode points to a double-indirect block, containing 128 indirect block numbers, and the 13<sup>th</sup> block number points to a triple-indirect block, containing 128 double-indirect block numbers. The inode contains a 4B file-size field. The block size is 512B.

Which of the following adjustments will allow files larger than the current 1GB limit to be stored?

- (a) Increase just the file-size field in the inode from a 32-bit to a 64-bit value

YES or NO. Give reason.

Doing so does not change the block size or the number of blocks that hold the pointers. Therefore this step will not achieve the objective

- (b) Increase just the number of bytes per block from 512 to 2048 bytes

YES or NO. Give reason.

Increasing the size of the block buys us more pointers for indirect connection blocks hence it will achieve the objective

- (c) Reformat the disk to increase the number of inodes allocated to the inode table

YES or NO. Give reason.

The number of inodes have no bearing on the size of a file.

- (d) Replace one of the direct block numbers in each inode with an additional triple-indirect block number

YES or NO. Give reason.

Any added indirection directly contributes to additional pointers, and hence additional blocks of data. So this step achieves our objective.

**Question 3b. [3 points] Answer these three questions by selecting the best response for each.**

**3b.1 "The target coordinates of a missile should not be improperly modified." This is an example of which security objective?**

- (A) Confidentiality
- (B)** Data integrity
- (C) Origin integrity
- (D) Availability
- (E) Authentication

**3b.2 "Paychecks should be printed and delivered on time." This is an example of which security objective?**

- (A) Confidentiality
- (B) Data integrity
- (C) Origin integrity
- (D)** Availability
- (E) Authentication

**3b.3 Which of the following is INCORRECT about security objectives?**

- (A) Security objectives include confidentiality, integrity, and availability.
- (B) Authentication is a critical security mechanism to ensure the objective of integrity.
- (C) Denial of service attacks are big threats to availability.
- (D) Confidentiality is about keeping secrets.
- (E)** We usually do not care about availability, since once confidentiality and integrity are assured, availability doesn't matter.

**Question 4. [10 points] Circle the best answer for each of the 10 multiple choice questions below.**  
*If in doubt about your answer, use the space below to explain in less than 2 sentences.*

- A Unix directory contains
  - a. Names and inode number of files
  - b. Names and sizes of files
- Which of the following events may not generate a signal?
  - a. Division by zero
  - b. A new connection arrives on a listening socket
  - c. A write is attempted on a disconnected socket
  - d. NULL is dereferenced
  - e. A process whose parent has already terminated exits
- Suppose that the kernel delivers two SIGCHLD signals to the parent while the parent is not scheduled. When the kernel finally schedules the parent, how many times will the SIGCHLD handler be called?
  - a. None, because sending multiple signals will always crash the program.
  - b. Exactly once, because signals are not queued.
  - c. Exactly twice, because signals are queued.
  - d. More than twice, depending on how the handler is installed
- Which of the following system calls can fail due to a network failure?
  - a. socket(...)
  - b. listen(...)
  - c. bind(...)
  - d. gethostbyname(...)
- A server can handle several requests at once by
  - a. using fork to create a new process for each request
  - b. using socket to create a new socket for each request
- To prevent race conditions, threads should use
  - a. signals
  - b. mutex objects
- The input redirection symbol "<" is processed by
  - a. the program that reads the data
  - b. the shell
- The listen system call is used to
  - a. receive data from a client
  - b. enable incoming connections to a socket
- A pipe can be used to transfer data between
  - a. only two processes
  - b. any number of processes
- When setting up a pipe to a child, the order of system calls is
  - a. the parent calls pipe then calls fork
  - b. the parent calls fork, then both processes call pipe

**Question 5. Thread Synchronization [10 points, 4+6 points respectively]**

- a. **Semaphore [4 points]** Consider three concurrently executing threads in the same process using two semaphores  $s_1$  and  $s_2$ . Assume  $s_1$  has been initialized to 1, while  $s_2$  has been initialized to 0. What are all possible values of the global variable  $x$ , initialized to 0, after all three threads have terminated? Show your work in reaching the answer.

/\* thread A \*/

P(&s2);

P(&s1);

$x = x * 2;$

V(&s1);

→ ABC order does not exist

→ CAB will result in

$$n = 0 + 3; \quad x = 3 * 2; \quad n = 6 * 6 \\ \therefore x = 36$$

/\* thread B \*/

P(&s1);

$x = x * x;$

V(&s1);

→ CBA will result in

$$n = 0 + 3; \quad x = 3 * 3; \quad x = 9 * 2 \\ \therefore x = 18$$

/\* thread C \*/

P(&s1);

$x = x + 3;$

V(&s2);

V(&s1);

→ BCA will result in

$$n = 0 * 0; \quad x = 0 + 3; \quad x = 3 * 2 \\ \therefore x = 6$$

So all possible values are :

$$x = 6, \quad x = 18, \quad x = 36$$

~~SE2 P6~~

Rubric :

- By default, assumption is that each semaphore is a binary semaphore. If someone assumed counting semaphores, you may grade them on their merit. It is acceptable.
- +4 points for all correct answers (+1.33 points per value)

- b. Reader-Writer [6 points]: There is 1 thread that writes to a file, and a number of other threads that can simultaneously read the file, but never the readers and the write together (e.g., at one time, 3 reader threads are running, at another time the writer thread is running). Write pseudo code for both the Reader() and Writer() functions. Remember that there is no data dependency between the readers and the writer (i.e., it is not a producer-consumer problem), the only condition is that they do not happen at the same time. Hint: The first reader to get access waits until the ongoing writer (if any) finishes, performs the read operation and leaves. The last reader, just before leaving, should give control/access back to the writer (if any) waiting. Think about the semaphores and mutexes you will need to solve the problem as a starting point.

```

W. // globals
    mut = Semaphore(1)
    wrt = Semaphore(1)
    readcount = 0;

    // writer function, just needs to wait on wrt sema.
    Writer()
    {
        while(true){
            wrt.P()

            // perform write
            wrt.V()

        } // end while loop
    }

    Reader()
    {
        while(true){
            mut.P()
            readcount++

            if(readcount == 1)
            {
                wrt.P()
            }

            mut.V()

            // perform the
            // read operation

            mut.P()
            readcount--
            if(readcount == 0)
            {
                wrt.V()
            }

            mut.V()
        }
    }
}

```

### Question 6. Signals, IPC, and Networking

- a. Signals [4 points] Consider the following C program

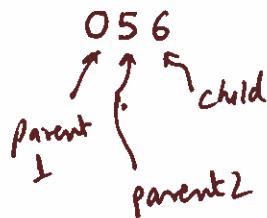
```
int counter = 0;
void handler1(int sig) {
    printf("%d", counter);
    kill(getpid(), SIGUSR2);
}
void handler2(int sig) {
    counter = 5;
    printf("%d", counter);
}
int main(int argc, char *argv[])
{
    int pid;
    signal(SIGUSR1, handler1);
    signal(SIGUSR2, handler2);
    if ((pid = fork()) == 0) {
        kill(pid, SIGUSR1);
    } else {
        counter++;
        printf("%d", counter);
    }
    return 0;
}
```

→ This question stands to earn up to +2 bonus points for the extra work! See rubric below.

Using the following assumptions, list possible outputs of the code:

- All processes run to completion and no system calls will fail
- `printf()` is atomic and calls `fflush(stdout)` after printing argument(s) but before returning

There are several possible outcomes here. One such example is



- Similarly other cases of reordered execution can be constructed.
- 1, 01, 11, 015, 055, 115, 155

Rubric { +2 points for 1 correct answer  
+4 points for at least 4 correct outputs  
Bonus of +2 point for all possible values

**6b. IPC [3 points, 0.5 points each]**

---

Socket connections are similar to pipe connections in that

- both use their file descriptors as addresses
- processes transfer data through both with the read and write system calls

A server program uses the bind system call to

- lock the socket in memory
- attach an address to its socket

A client program makes a connection to a server program by calling

- socket
- connect

The select system call is used to

- block on input from several file descriptors at once
- connect two file descriptors

The select system call returns 0 if

- it detects end of file on a file descriptor
- the timeout value is reached

A process cannot open a named pipe for reading until

- a process writes data into the pipe
  - a process opens the pipe for writing
-

## Question 6. Signals and Networking

**6b. Networking [3 points, 0.5 points each] Multiple Choice Questions. Write the LETTER for BEST CHOICE in [ ] appended to the problem statement**

Final Test  
CSCE 313 Fall 2018  
(150 points)

[This exam has 10 questions spanning 8 pages]

Student Name: .....

UIN:.....

I have adhered to Aggie Code of Honor

Signature:.....

**Question 1 [30 points] Circle True (T) or False (F) for the following statements**

- a) T F Threads have less context switch overhead compared to that of processes
- b) T F The dominant factor in process context switch is due to Memory/IO-state switch overhead
- c) T F **Rotational disks are slow primarily due to their rotational latency**
- d) T F The function **bind()** on the server side attaches the process to a port
- e) T F Each lock has a separate wait queue
- f) T F **connect() function is blocking, while accept() function is non-blocking**
- g) T F **Blocking a signal means that the process ignores the signal upon receiving it**
- h) T F While a signal is being handled, and there are more instances of the same signal, then one instance of them is being pending
- i) T F A soft link keeps the new name in the directory entry and the original file name in data block
- j) T F All hard links of a file share the same inode
- k) T F **A semaphore object should have a getvalue() (i.e., returns the current value of the counter) function, because it is required for using the semaphore correctly**
- l) T F A **test&set()**-based lock can be used completely in user mode – no elevation to Kernel mode is necessary
- m) T F With a **test&set()**-based lock, there is always some sort of busy-wait
- n) T F **Message Queues have less memory overhead compared to Shared Memory**
- o) T F For some signals, the Kernel works as the source and for others it works as the router
- p) T F **SIGINT cannot be handled or ignored**
- q) T F **SIGCHLD** is ignored by default
- r) T F A directory entry contains both the file name and the inode number
- s) T F **The listen() function attaches a server process to a port number**
- t) T F **getaddrinfo()** function is used to look up the IP address from domain name
- u) T F Domain Name System (DNS) is a distributed database of domain name to IP address mappings
- v) T F No data is written to the disk file when exchanging data through a FIFO
- w) T F **shmat() function creates a shared memory IPC object**
- x) T F **accept() function returns a slave socket that is ready for communication (i.e., you do not need to bind/listen on it)**
- y) T F **When a TCP server (e.g., web server) needs to send data back to the TCP client, it writes on to the master socket**
- z) T F There can be delay between the time a signal is generated and the time when it is delivered to the recipient process
- aa) T F The default action for most signals is to exit the process
- bb) T F UDP is connectionless while TCP is connection-based
- cc) T F DNS can be used by busy servers as a way for load balancing
- dd) T F A TCP connection is reliable; if packets are lost in route, they are recovered through retransmission

**Question 2 [5 pts]:** What are the main problems regarding the following Interrupt-based lock implementation? [You need to mention at least 2 problems.]

<pre>int value = 0; Acquire() {     disable interrupts;     if (value == 1) {         put thread on wait-queue;         go to sleep()     } else         value = 1;     enable interrupts; }</pre>	<pre>Release() {     disable interrupts;     if anyone on wait queue {         take thread off wait-queue         Place on ready queue;     } else {         value = 0;     }     enable interrupts; }</pre>
--	--

Answer:

1. Enabling/disabling Interrupts will require elevation of privileges
2. Interrupts can be still disabled for a long time, making the system less responsive
3. Problematic in multi-processor because need to disable interrupt in all processors through off-chip message passing which is very time-consuming

**Question 3 [10 points]:** The following is the definition of a an atomic test&set() instruction. Implement a lock using this instruction. More specifically, write the Acquire() and Release() functions also declare the necessary global variables

```
int test&set (&address) {
    result = M[address];
    M[address] = 1;
    return result;
}
```

Answer:

<pre>int value = 0; // Free  Acquire() {     while (test&amp;set(value)); }  Release() {     value = 0; }</pre>
---

**Question 4 [5 points]:** Describe what happens with the file system in terms of inode table, disk blocks, and directory entry when you create a 10KB file named “test”. Assume 4KB data blocks.

1. First find a free inode
2. Then find 3 blocks to keep the data
3. Copy the data to the disk blocks
4. Keep these block numbers in the inode
5. Make a directory entry where you keep the filename “test” and the inode number

**Question 5 [10 points]:** Write a program to list the file/directory names in a directory.

```
void listdir (string dirname){  
    struct dirent * direntp;  
    DIR * dirp;  
    if ((dirp = opendir(dirname.c_str())) == NULL) {  
        perror("Failed to open directory");  
        return 1;  
    }  
    while ((direntp = readdir(dirp)) != NULL)  
        printf("%s\n", direntp->d_name);  
  
    closedir(dirp);  
    return 0;  
}
```

**Question 6 [10 points]:** Write a piece of a Producer and a Consumer thread with a single buffer (a buffer with just 1 slot) between them using minimum number of semaphores. More specifically, in the following, add minimum number of semaphores and then fill out Wait() and Done() functions of both the Producer and the Consumer. No need to fill out the ProduceItem() and ConsumeItem() functions.

<pre>void Producer () {     while (true) {         ProducerWait();         ProduceItem();         ProducerDone();     } }</pre>	<pre>void Consumer () {     while (true) {         ConsumerWait();         ConsumeItem();         ConsumerDone();     } }</pre>
---	---

**Answer:**

Sempahore full (0), empty(1);

<pre>ProducerWait () {     empty.P(); }  ProducerDone () {     full.V(); }</pre>	<pre>ConsumerWait () {     full.P(); }  ConsumerDone () {     empty.V(); }</pre>
--	--

**Question 7 [20 points]:** Assume that you are writing a piece of code that is part of the Gmail App and it occasionally prints messages to keep the users informed/entertained during the time the mails take a long time to load over slow Internet connections. The function

`fetch_emails(userid)` is given to you and it fetches emails for the user but can take a random amount of time. However, this is a blocking function that does not return unless emails are fetched completely. During this time, you need to print the following messages in the browser status bar (by calling the given function `status_update(message)`) **every second** based on how much time has elapsed since `fetch_emails(userid)` function was called:

Elapsed Time (sec)	Message
[0,5)	Working
[5,10)	Still working
[10,15)	Still working, apologies for the delay
[15,20)	Well, it's embarrassing!! Still working on it.

Important things to consider:

- a. You must update status every second. Printing the same message again is fine.
- b. You **cannot** use multiple threads or call sleep function – it must use signal handling
- c. Assume that **SIG\_ALRM** is being generated every second – you do not need to set it up. You just need to decide what to do with the **SIG\_ALRM**
- d. The function `timediff()` is also given and it returns Sseconds since `fetch_emails()` invocation
- e. Your solution must have a `main()` function and any necessary supporting function(s). Do not rewrite the given functions `fetch_emails`, `status_update`, and `timediff`.

```

string msgs [] = {"Working", "Still working", "Still working,
apologies for the delay", "Well, it's embarrassing!! Still
working on it"};
}

void handler (int sig){
    int td = timediff();
    status_update (msgs [td/5]);
}

int main (){
    signal (SIG_ALRM, handler);
    fetch_email (userid);
}

```

**Question 8 [20 pts]:** You are to solve a variant of the classical Reader-Writer problem, where there are exactly **R reader** threads and exactly **W writer** threads trying to do their read/write operations. However, there is a producer-consumer relation here – W worker threads must finish first (i.e., produce the data) and after that, R reader threads will run (i.e., consume the data) and the whole thing will repeat. So, there is no overlap between the writers and the readers. However, all W writers run together and then all R readers run together.

You have to fill out the following 4 functions: **ReaderWait()** , **ReaderDone()** , **WriterWait()** , **WriterDone()** . Declare additional Semaphores if you need. However, using unnecessary semaphores will result in lost points.

<pre>Reader() {     do{         ReaderWait();         PerformReadOperation();         ReaderDone();     }while(1); }</pre>	<pre>Writer() {     do{         WriterWait();         PerformaWriteOperation();         WriterDone();     }while (1); }</pre>
--	---

Answer:

<pre>Semaphore writers_done(0); Semaphore readers_done(W); Semaphore mtx (1); int nreaders = 0, nwrites = 0;  WriterWait () {     readers_done.P(); }  WriterDone () {     mtx.P();     nwrites++;     if (nwrites == W) {         nwrites = 0;         for (int i=0;i&lt;R;i++)             writers_done.V();     }     mtx.V(); }</pre>	<pre>ReaderWait () {     writers_done.P(); }  ReaderDone () {     mtx.P();     nreaders++;     if (nreaders == R) {         for (int i=0;i&lt;W;i++)             readers_done.V();     }     mtx.V(); }</pre>
---	---

**Question 9 [20 points]:** Consider the program below and answer the following questions with proper explanation.

- What is the output? How much time does the program take to run? [10 points]
- What is the output with line 5 commented? How much time will it take now? [10 points]

```
1 void signal_handler (int signo){  
2     printf ("Got SIGUSR1\n");  
3 }  
4 int main () {  
5     signal (SIGUSR1, signal_handler); //comment out for b)  
6     int pid = fork ();  
7     if (pid == 0){// child process  
8         for (int i=0; i<5; i++){  
9             kill (getppid(), SIGUSR1);  
10            sleep (1);  
11        }  
12    }else{ // parent process  
13        wait (0);  
14    }  
15 }
```

- Got SIGUSR1 5 times, will take 5 seconds
- Nothing is printed because the first SIGUSR1 kills the main process

**Question 10 [20 pts]:** Write pseudocode to set up a multi-threaded TCP server that is capable of accepting and handling multiple client connections at a time. You need to mention all the major POSIX functions in your code and explain what they do or why they are necessary. This explanation can be given either separately or in comments with code. Note that some of these functions are called only once (in the beginning) to setup the server, and some are called repeatedly in a loop. Please mention all of them in your pseudocode.

Answer:

```
getaddrinfo();  
int master_socket = socket();  
bind(master_socket, portno); // binding to the port  
listen(master_socket, 20); // setting up a wait queue  
while (true){  
    if slave_socket = accept (master_socket);  
    pthread_create (tid, 0, handler_function, &slave_socket);  
}
```

Final Test  
CSCE 313 Spring 2018  
(140 points)

[This exam has 9 questions spanning 7 pages]

Date: May 4, 2018

Student Name: ..... **KEY** .....

UIN:.....

I have adhered to Aggie Code of Honor

Signature:.....

**Question 1 [30 points] Circle True (T) or False (F) for the following statements**

- a) T F Threads have less context switch overhead compared to that of processes
- b) T F The dominant factor in process context switch is due to Memory/IO-state switch overhead
- c) T F Rotational disks are slow primarily due to their rotational latency
- d) T F Making a symbolic link increases the original inode's reference/link count
- e) T F The function **bind()** on the server side attaches the process to a port
- f) T F Each lock has a separate wait queue
- g) T F **connect()** function is blocking, while **accept()** function is non-blocking
- h) T F Blocking a signal means that the process ignores the signal upon receiving it
- i) T F While a signal is being handled, and there are more instances of the same signal, then one instance of them is being pending
- j) T F A soft link keeps the new name in the directory entry and the original file name in data block
- k) T F All hard links of a file share the same inode
- l) T F A **test&set()**-based lock can be used completely in user mode – no elevation to Kernel mode is necessary
- m) T F With a **test&set()**-based lock, there is always some sort of busy-wait
- n) T F Message Queues have less memory overhead compared to Shared Memory
- o) T F For some signals, the Kernel works as the source and for others it works as the router
- p) T F **SIGINT** cannot be handled or ignored
- q) T F **SIGCHLD** is ignored by default
- r) T F A directory entry contains both the file name and the inode number
- s) T F The **listen()** function attaches a server process to a port number
- t) T F **getaddrinfo()** function is used to look up the IP address from domain name
- u) T F Domain Name System (DNS) is a distributed database of domain name to IP address mappings
- v) T F No data is written to the disk file when exchanging data through a FIFO
- w) T F **shmat()** function creates a shared memory IPC object
- x) T F **accept()** function returns a slave socket that is ready for communication (i.e., you do not need to bind/listen on it)
- y) T F When a TCP server (e.g., web server) needs to send data back to the TCP client, it writes on to the master socket
- z) T F There can be delay between the time a signal is generated and the time when it is delivered to the recipient process
- aa) T F The default action for most signals is to exit the process
- bb) T F UDP is connectionless while TCP is connection based
- cc) T F DNS can be used by busy servers as a way for load balancing
- dd) T F A TCP connection is reliable; if packets are lost in route, they are recovered through retransmission

**Question 2 [10 points]:** Assuming initially  $x = 0$ , what are the possible values of the  $x$  after running 3 instances of the following thread function (i.e., each increment  $x$  by 1)? Explain your answer with machine level instructions `lw` (loads an integer), `add`, `sw` (stores an integer).

```
ThreadFunc () {  
    x++;  
}  
  
x = 1, 2, 3
```

**Question 3 [5 points]:** Describe what happens with the file system in terms of inode table, disk blocks, and directory entry when you create a 10KB file named “test”. Assume 4KB data blocks.

**Answer:**

1. First find a free inode
2. Then find 3 blocks to keep the data
3. Copy the data to the disk blocks
4. Keep these block numbers in the inode
5. Make a directory entry where you keep the filename “test” and the inode number

**Question 4 [5 points]:** The unix “cat filename” command reads the content of the filename and prints that to the standard output. Describe how this works. Hint: Start with searching the filename in the current directory and end with reading the disk blocks associated with the file.

**Answer:**

1. Lookup file “filename” in the current directory and find the corresponding inode no
2. Index into the inode table with the number and locate the inode entry
3. From the inode make sure that you have permissions, if not do not go any further
4. If otherwise, locate the disk blocks from the inode, read them in order and print out the content to the standard output

**Question 5 [10 points]:** Write code for a BoundedBuffer where you do not care about underflow; rather the **only concern is overflow** of the buffer. This can happen when the producers are much faster than the consumers are. As a result, the buffer never depletes. However, the buffer might grow indefinitely unless you avoid that in your code. **Fill out the constructor, push() and pop() functions below.** Constraints: You must use the minimum number of Semaphores/Mutexes possible (otherwise, you could still check for underflow, which would not hurt, right?), and you cannot have unnecessary checks for stopping underflow.

<pre>class BoundedBuffer{ private:     /* add minimum semaphores */ queue&lt;string&gt; q; public: BoundedBuffer(int capacity){  }</pre>	<pre>void push (string data){ }  string pop () { }  };</pre>
--	--

**Answer:**

```
class BoundedBuffer{
private:
    Semaphore*      empty;
    Semaphore*      mutex;
    queue<string> q;
public:
BoundedBuffer(int capacity){
    empty = new Semaphore (capacity);
    mutex = new Semaphore (1);
}
void push (string data){
    empty.P();
    mutex.P();
    q.push (data);
    mutex.V();
}
string pop () {
    mutex.P();
    string data = q.pop ();
    mutex.V();
    empty.P();
    return data;
}
};
```

**Question 6 [20 points]** A Linux file system has inodes that are each 256 bytes in size and each include 24 direct, 1 single indirect, 1 double indirect, and 1 triple indirect pointer. Now, the disk is formatted using 8KB data blocks, where each pointer is 8 bytes. Compute the following information for this file system. Explain your answer.

- (a) Maximum file size
- (b) Overhead (non-data information) in bytes for keeping a maximum size file

**Answer:**

- (a)** Max file size =

**Question 7 [20 points]:** Assume that you are writing a piece of code that is part of the Gmail App and it occasionally prints messages to keep the users informed/entertained during the time the mails take a long time to load over slow Internet connections. The function `fetch_emails(userid)` is given to you and it fetches emails for the user but can take a random amount of time. However, this is a blocking function that does not return unless emails are fetched completely. During this time, you need to print the following messages in the browser status bar (by calling the given function `status_update(message)`) **every second** based on how much time has elapsed since `fetch_emails(userid)` function was called:

Elapsed Time (sec)	Message
[0,5)	Working
[5,10)	Still working
[10,15)	Still working, apologies for the delay
[15,20)	Well, it's embarrassing!! Still working on it.

Important things to consider:

- a. You must update status every second. Printing the same message again is fine.
- b. You **cannot** use multiple threads or call sleep function – it must use signal handling
- c. Assume that **SIG\_ALRM** is being generated every second – you do not need to set it up. You just need to decide what to do with the **SIG\_ALRM**
- d. The function `timediff()` is also given and it returns Sseconds since `fetch_emails()` invocation
- e. Your solution must have a `main()` function and any necessary supporting function(s). Do not rewrite the given functions `fetch_emails`, `status_update`, and `timediff`.

Answer: See 2019 Spring Final KEY

**Question 8 [20 pts]:** You are to solve a variant of the classical Reader-Writer problem, where there are exactly **R reader** threads and exactly **W writer** threads trying to do their read/write operations. However, there is a producer-consumer relation here – the W worker threads must finish first (i.e., produce the data) and after that the R reader threads will run (i.e., consume the data) and the whole thing will repeat. So, there is no overlap between the writers and the readers. However, all W writers run together and then all R readers run together.

You have to fill out the following 4 functions: **ReaderWait()** , **ReaderDone()** , **WriterWait()** , **WriterDone()** . Declare additional Semaphores if you need. However, using unnecessary semaphores will result in lost points.

Reader () { <b>do{</b> <b>ReaderWait();</b> PerformReadOperation (); <b>ReaderDone();</b> <b>}while(1);</b> }	Writer () { <b>do{</b> <b>WriterWait();</b> PerformaWriteOperation (); <b>WriterDone();</b> <b>}while (1);</b> }
---	--

**Question 9 [20 points]:** Consider the program below and answer the following questions with proper explanation.

- a) What is the output? How much time does the program take to run?[10 points]
- b) What is the output with line 5 commented? How much time will it take now? [10 points]

```
1 void signal_handler (int signo){  
2     printf ("Got SIGUSR1\n");  
3 }  
4 int main () {  
5     signal (SIGUSR1, signal_handler); //comment out for b)  
6     int pid = fork ();  
7     if (pid == 0){// child process  
8         for (int i=0; i<5; i++){  
9             kill(getppid(), SIGUSR1);  
10            sleep (1);  
11        }  
12    }else{ // parent process  
13        wait(0);  
14    }  
15 }
```

**Answer:** See 2019 Spring Final Key for the answer to this question

Final Test  
CSCE 313 Spring 2019  
(150 points)

[This exam has 10 questions spanning 8 pages]

Student Name: .....

UIN:.....

I have adhered to Aggie Code of Honor

Signature:.....

**Question 1 [30 points] Circle True (T) or False (F) for the following statements**

- a) T F Threads have less context switch overhead compared to that of processes
- b) T F The dominant factor in process context switch is due to Memory/IO-state switch overhead
- c) T F **Rotational disks are slow primarily due to their rotational latency**
- d) T F The function **bind()** on the server side attaches the process to a port
- e) T F Each lock has a separate wait queue
- f) T F **connect() function is blocking, while accept() function is non-blocking**
- g) T F **Blocking a signal means that the process ignores the signal upon receiving it**
- h) T F While a signal is being handled, and there are more instances of the same signal, then one instance of them is being pending
- i) T F A soft link keeps the new name in the directory entry and the original file name in data block
- j) T F All hard links of a file share the same inode
- k) T F **A semaphore object should have a getvalue() (i.e., returns the current value of the counter) function, because it is required for using the semaphore correctly**
- l) T F A **test&set()**-based lock can be used completely in user mode – no elevation to Kernel mode is necessary
- m) T F With a **test&set()**-based lock, there is always some sort of busy-wait
- n) T F **Message Queues have less memory overhead compared to Shared Memory**
- o) T F For some signals, the Kernel works as the source and for others it works as the router
- p) T F **SIGINT cannot be handled or ignored**
- q) T F **SIGCHLD is ignored by default**
- r) T F A directory entry contains both the file name and the inode number
- s) T F The **listen()** function attaches a server process to a port number
- t) T F **getaddrinfo()** function is used to look up the IP address from domain name
- u) T F Domain Name System (DNS) is a distributed database of domain name to IP address mappings
- v) T F No data is written to the disk file when exchanging data through a FIFO
- w) T F **mmap() function creates a shared memory IPC object**
- x) T F **accept() function returns a slave socket that is ready for communication (i.e., you do not need to bind/listen on it)**
- y) T F **When a TCP server (e.g., web server) needs to send data back to the TCP client, it writes on to the master socket**
- z) T F There can be delay between the time a signal is generated and the time when it is delivered to the recipient process
- aa) T F The default action for most signals is to exit the process
- bb) T F UDP is connectionless while TCP is connection-based
- cc) T F **kill() function is used only to kill a process**
- dd) T F Shared Memory IPC has less memory overhead compared to FIFO and Message Queues

**Question 2 [5 pts]:** What are the main problems regarding the following Interrupt-based lock implementation? You need to mention at least 2 problems.

<pre> int value = 0; Acquire() {     disable interrupts;     if (value == 1) {         put thread on wait-queue;         go to sleep()     } else         value = 1;     enable interrupts; } </pre>	<pre> Release() {     disable interrupts;     if anyone on wait queue {         take thread off wait-queue         Place on ready queue;     } else {         value = 0;     }     enable interrupts; } </pre>
--	--

Answer:

1. Enabling/disabling Interrupts will require elevation of privileges
2. Interrupts can be still disabled for a long time, making the system less responsive
3. Problematic in multi-processor because need to disable interrupt in all processors through off-chip message passing which is very time-consuming

**Question 3 [10 points]:** The following is the definition of a an atomic test&set() instruction. Implement a lock using this instruction. More specifically, write the Acquire() and Release() functions also declare the necessary global variables

<pre> int test&amp;set (&amp;address) {     result = M[address];     M[address] = 1;     return result; }  int value = 0; // Free </pre>	<pre> Acquire() {     // Short busy-wait time     while (test&amp;set(value)); }  Release() {     value = 0; } </pre>	<pre> int guard = 0; //protects "value"  int value = FREE;  Acquire() {     // Short busy-wait time     while (test&amp;set(guard));     if (value == BUSY) {         put thread on wait queue;         go to sleep() &amp; guard = 0;     } else {         value = BUSY;         guard = 0;     } }  Release() {     // Short busy-wait time     while (test&amp;set(guard));     if (anyone on wait queue) {         take thread off wait queue         Place on ready queue;     } else {         value = FREE;     }     guard = 0; } </pre>
--	---	--

**Question 4 [5 points]:** Describe what happens with the file system in terms of inode table, disk blocks, and directory entry when you create a 10KB file named “test”. Assume 4KB data blocks.

- 1) Find a free inode
- 2) Find 3 free disk blocks and copy content there
- 3) Put the block no's in the inode
- 4) Add file name and inode no to directory entry

**Question 5 [10 points]:** The following code will create a Zombie child process because the child process is terminated and the parent process is busy in a loop without calling `wait()` function. Now, modify this program so that no Zombie process is created. Hint: do this through handling `SIGCHLD` signal. The parent process cannot call `wait()` directly in the `main()` (inside the signal handler is fine) and it still has to go to the infinite while loop. You can add helper functions.

```
int main() {
    if (fork()== 0) // child process
        exit(0);
    else // parent process
        while (true);
}
```

**Answer:**

```
int handler(int sno){
    wait(0);
}

int main(){
    signal (SIGCHLD, handler);
    if (fork()== 0) // child process
        exit(0);
    else // parent process
    {
        //signal (SIGCHLD, handler); here is also fine
        while (true);
    }
}
```

**Question 6 [10 points]:** In the following, add minimum number of semaphores and then define Producer/ConsumerWait() and Producer/ConsumerDone() functions. No need to fill out the ProduceItem() and ConsumeItem() functions.

<pre>void Producer (){     while (true){         ProducerWait();         ProduceItem();         ProducerDone();     } }</pre>	<pre>void Consumer (){     while (true){         ConsumerWait();         ConsumeItem();         ConsumerDone();     } }</pre>
---	---

```
////////// Solution ///////////  
Semaphore full (0), empty(1);  
  
PWait(){  
    empty.P(); // wait for an empty slot  
}  
PDone(){  
    full.V(); // unleash the consumers  
}  
  
CWait(){  
    full.P(); // only this one needs the full, others piggyback  
}  
  
CDone(){  
    empty.V(); // this will unlease exactly M producers  
}
```

**Question 7 [20 pts]:** You are to solve a variant of the classical Produce-Consumer problem (as in Question 6) where you let **M producer** threads run first and then **N consumer** threads and then, the whole thing will repeat. Note that there is a single buffer that is being prepared by all M producers (for instance, M threads loading M different portions of a single web page using M HTTP connections to the server, then you are using N parsing threads to parse the web page). However, there is no race condition between the producers, because they work on disjoint portions of the buffer. Similarly, the consumer threads do NOT race with each other either. Use the skeleton in Question 6 as the given code, add necessary semaphores and then define the 4 functions: Producer/ConsumerWait(), Producer/ConsumerDone().

```
////////// Solution ///////////
Semaphore full (0), empty(M);
int pcount = 0, ccount = 0; // number of producers/consumers done so far
Mutex pcm, ccm; // mutexes to protect the above counts

PWait(){
    empty.P(); // wait for an empty slot
}
PDone(){
    pcm.Lock();
    pcount++;
    if (pcount == M){
        pcount = 0; // reset counter
        for (int i=0; i<N; i++)
            full.V(); // unleash the consumers
    }
    pcm.Unlock();
}

CWait(){
    full.P(); // only this one needs the full, others piggyback
}

CDone(){
    ccm.Lock();
    ccount++;
    if (ccount == N){ //the last one in a caravan, this needs to wake up all
M prods
        for (int i=0; i<M; i++)
            empty.V(); // this will unleash exactly M producers
    }
    ccm.Unlock();
}
```

**Question 8 [20 points]:** Consider the program below and answer the following questions with proper explanation.

- a) What is the output? How much time does the program take to run?[10 points]
- b) What is the output with line 5 commented? How much time will it take now? [10 points]

```
1 void signal_handler (int signo){  
2     printf ("Got SIGUSR1\n");  
3 }  
4 int main () {  
5     signal (SIGUSR1, signal_handler); //comment out for  
b)  
6     int pid = fork ();  
7     if (pid == 0){// child process  
8         for (int i=0; i<5; i++){  
9             kill(getppid(), SIGUSR1);  
10            sleep (1);  
11        }  
12    }else{ // parent process  
13        wait(0);  
14    }  
15}
```

- a) Got SIGUSR1 5 times, will take 5 seconds
- b) Nothing is printed because the first SIGUSR1 kills the main process

**Question 9 [20 pts]:** A Linux file system has inodes that are each 256 bytes in size and each include 24 direct, 1 single indirect, 1 double indirect, and 1 triple indirect pointers. Now, the disk is formatted using 8KB data blocks, where each pointer is 8 bytes. Compute the following information for this file system and explain your answer. [Recap: 1024 bytes = 1KB, 1024KB = 1MB, 1024MB = 1GB, 1024GB = 1TB]

- (a) Maximum file size possible in this file system
- (b) Overhead (non-data information) in bytes for keeping a maximum size file. Ignore the 256 bytes for storing each inode. Just consider the pointer blocks.

$$\text{Filesize} = 24 \times 8\text{KB} + 1024 \times 8\text{KB} + 1024^2 \times 8\text{KB} + 1024^3 \times 8\text{KB}$$

$$\text{Overhead} = 0 + 8\text{KB} + (1 + 1024) 8\text{KB} + (1 + 1024 + 1024^2) 8\text{KB}$$

**Question 10 [20 points]:** Assume that you are writing a piece of code that is part of the Gmail App and it occasionally prints messages to keep the users informed/entertained during the time the mails take a long time to load over slow Internet connections. The function `fetch_emails(userid)` is given to you and it fetches emails for the user but can take a random amount of time. However, this is a blocking function that does not return unless emails are fetched completely. During this time, you need to print the following messages in the browser status bar (by calling the given function `status_update(message)`) **every second** based on how much time has elapsed since `fetch_emails(userid)` function was called:

Elapsed Time (sec)	Message
[0,5)	Working
[5,10)	Still working
[10,15)	Still working, apologies for the delay
15 or more	Well, it's embarrassing!! Still working on it.

Important things to consider:

- a. You must update status every second. Printing the same message again is fine.
- b. Assume that `SIGALRM` is being generated every second – you do not need to set it up. You just need to decide what to do with the `SIGALRM`
- c. The function `timediff()` is also given and it returns seconds since `fetch_emails()` invocation
- d. Your solution must have a `main()` function and any necessary variables and supporting function(s).

Answer:

```
string msgs [] = {"Working", "Still working", "Still working,
apologies for the delay", "Well, it's embarrassing!! Still
working on it"};
}

void handler (int sig){
    int td = timediff();
    status_update (msgs [td/5]);
}

int main (){
    signal (SIGALRM, handler);
    fetch_email (userid);
}
```

Name:..... UIN:.....

Score:  out of Total 100

**Question 1 [10 pts]:** The following is the Producer(.) function in a BoundedBuffer implementation. What is the purpose of the mutex in the following? Can we do without the mutex? In what circumstances?

```
Producer(item) {  
    emptySlots.P();  
    mutex.P();  
    Enqueue(item);  
    mutex.V();  
    fullSlots.V();  
}
```

*Answer: Multiple producer and consumer threads can be in the critical section modifying the queue (i.e., enqueueing or dequeuing). The mutex makes sure that those modifications happen only 1 at a time.*

**Question 2 [10 pts]:** The following is the Producer(.) function in a BoundedBuffer implementation. Can we change the order of the first 2 lines? Why or why not?

```
Producer(item) {  
    emptySlots.P();  
    mutex.P();  
    Enqueue(item);  
    mutex.V();  
    fullSlots.V();  
}
```

*Answer: We cannot because there is a potential scenario where this would result in a deadlock. The scenario is the following: When the queue is full to its capacity and the producer can lock the mutex, it is stuck on emptySlots.P() with the lock at hand. So, no consumer thread can go in to consume anything. As a result, the producer is stuck forever.*

**Question 3 [20 pts]:** If we run 5 instances of ThreadA() and 1 instance of ThreadB(), what can be the maximum number of threads active simultaneously in the Critical Section? The mutex is initially unlocked. Note that ThreadB() is buggy and mistakenly unlocks the mutex first instead of locking first. Explain your answer.

<pre>ThreadA() {     mutex.P()     /* Start Critical Section */     .....     /* End Critical Section */     mutex.V(); }</pre>	<pre>ThreadB() {     mutex.V()     /* Start Critical Section */     .....     /* End Critical Section */     mutex.P(); }</pre>
---	---

*The answer is 3. 1 ThreadA() locks and mutex =0. So no more ThreadA() can get it. However, a ThreadB() can get in fine and makes mutex=1, which can be consumed by another ThreadA().*

**Question 5 [25 pts]:** Consider a multithreaded web crawling and indexing program, which needs to first download a web page and then parse the HTML of that page to extract links and other useful information from it. The problem is both downloading a page and parsing it can be very slow depending on the content. Your goal is to make both these components as fast as possible. First, to speed up downloading, you delegate the task to **m** downloader threads, each with only a portion of the page to download. (Note that this is quite common in real life and a typical web browser does this all the time as long as the server supports this feature. Usually it is done through opening multiple TCP connection with the server and downloading equal sized chunk through each connection). The **M** chunks are downloaded to a single page buffer. Once all the chunks are downloaded into the buffer, you can then start parsing it. However, since you want to speed up parsing as well, you now use **n** parsing threads who again can parse the page independently, and together they take much less time.

By now, you probably see that **M** download threads are acting as Producers and **N** parser threads as Consumers. Additionally, note that the both downloader and parser threads come from a pool of **M** Producer threads and **N** Consumer threads where **M>m** and **N>n**. Out of many of these, you have to let exactly **m** Producer threads carry out the download and then exactly **n** consumer threads parse, and then the whole cycle will repeat. IOW, in each cycle, you are employing **m** out of **M** Producer worker threads (who are all eagerly waiting) to download the page simultaneously, and then **n** out of **N** Consumers are concurrently parsing the downloaded page. You cannot assume **m=M** or **n=N**. Assume that you can a function call **download(URL)** to download the page and **parse(chunk)** to parse a chunk of the page. No need to be any more specific/concrete than that. The main thing of interest is the Producer-Consumer relation.

Look at the given program **1PNC.cpp** that works for 1 Producer and **n** Consumer threads. Be sure to run the program first to see how it behaves. You need to

extend the program such that it works for m producers instead of just 1. Add necessary semaphores to the program. However, you will lose points if you add unnecessary Semaphores or Mutexes. To keep things simple, declare the mutexes as semaphores as well. You are given a fully implemented Semaphore.h class that you can use for Semaphores. Test your program to make sure that it is correct. In your submission directory, include a file called **Q5.cpp** that contains the correct program.

```
#include <iostream>
#include <thread>
#include <unistd.h>
#include <vector>
#include "Semaphore.h"
using namespace std;

#define NP 10
#define NC 5

int buffer = 0;
Semaphore consumerdone (NP);
Semaphore producerdone (0);
Semaphore mtx (1); // will use as mutex
int ncdone = 0; // number of consumers done consuming
int npdone = 0;

// each producer gets an id, which is pno
void producer_function (int pno){
    int count = 0; // each producer threads has its own count
    while (true){
        // do necessary wait
        consumerdone.P();      // wait for the buffer to be empty

        mtx.P();
        buffer++;
        cout << "Producer [" << pno << "] left buffer=" << buffer << endl;
        npdone++;
        if (npdone == NP){
            npdone = 0;
            for (int i=0; i<NC; i++)
                producerdone.V();
        }
        mtx.V();
    }
}
```

**Question 6 [15 pts]:** There are 3 sets of threads A, B, C. First 1 instance of A has to run, then 2 instances of B and then 1 instance of C, then the cycle repeats. This emulates a chain of producer-consumer relationship that we learned in class, but between multiple pairs of threads. Write code to run these set of threads.

Assumptions and Instructions: There are 100s of A, B, C threads trying to run. Write only the thread functions with proper wait and signal operation in terms of semaphores. You can use the necessary number of semaphores as long as you declare them in global and initialize them properly with correct values. The actual operations done by A, B and C does not really matter. Submit a separate C++ file called Q6.cpp that includes the solution.

```
#include <iostream>
#include <thread>
#include <unistd.h>
#include <vector>
#include "Semaphore.h"
using namespace std;

Semaphore Adone (0);
Semaphore Bdone (0);
Semaphore Cdone (1);
Semaphore mtx (1);
int bcount = 0;

void A (){
    while (true){
        Cdone.P();          // wait for the buffer to be empty
        cout << "A thread Running " << endl;

        // release twice to let 2 B threads run
        Adone.V();
        Adone.V();
    }
}

void B (){
    while (true){
        Adone.P();
        usleep (1000000);
        mtx.P();
        cout << "++++ B thread Running " << endl;
        bcount++;
        if (bcount == 2){
            bcount = 0;
            Bdone.V();
        }
        mtx.V();
    }
}
```

```
void C (){
    while (true){
        Bdone.P();
        cout << ">>>>>>>> C thread Running " << endl << endl;
        Cdone.V();
    }
}

int main (){
    vector<thread> As;
    vector<thread> Bs;
    vector<thread> Cs;

    for (int i=0; i< 20; i++){
        As.push_back (thread (A));
        Bs.push_back (thread (B));
        Cs.push_back (thread (C));
    }

    for (int i=0; i<As.size (); i++){
        As [i].join ();
        Bs [i].join ();
        Cs [i].join ();
    }
}
```

**Question 7 (20 pts):** Implement a Mutex using the atomic `swap(variable, register)` instruction in x86. Your mutex can use busy-spin. Note that you cannot access a register directly other than using this swap instruction.

```
int value = 0;
void Acquire (){
    while (true){
        int temp = 1;
        swap (temp, $reg); // make $reg = 1
        swap (value, $reg); // make value = 1

        //now check the old value of value
        swap (temp, $reg);
        if (temp == 0) // wait is over if value was 0
            break;
    }
}
```

```
void Release (){
    value = 0;
}

///////////
Register $reg;
Constructor(){
    Int temp = 0;
    Swap (temp, $reg);
}
void Acquire (){
    whlie (true){
        int temp = 1;
        swap (temp, $reg); // make $reg = 1
        if (temp == 0) // wait is over if value was 0
            break;
    }
}
void Release (){
    int temp = 0;
    swap (temp, $reg);
}
```

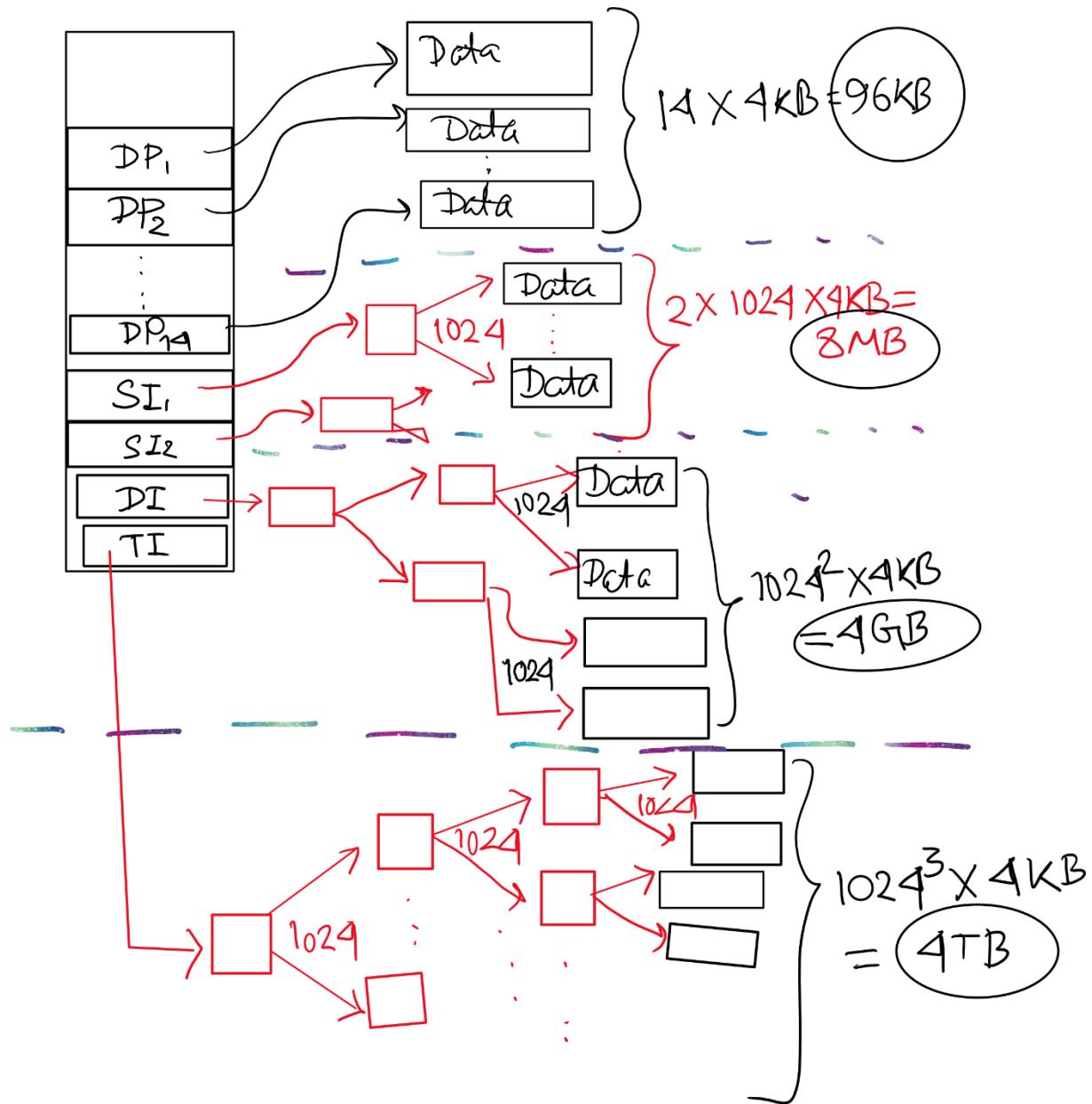
**TOTAL POINTS 100****TRUE/FALSE QUESTIONS – 1PT EACH [30 PTS] (ONLY THE FALSE ONES ARE CIRCLED)**

1. Shared memory IPC comes with built-in (kernel provided) synchronization
2. FIFOs persist without any processes connected to them
3. Shared memory and memory mapped files require 3 times memory overhead compared to FIFO and MQ
4. Pipes are supported by a First-In-First-Out bounded buffer given by the Kernel
5. POSIX message queues support separate priority levels for the messages
6. In POSIX message queues, the order of the messages is always FIFO without any exception
7. A unnamed pipe can be established only between processes in the same family tree
8. A unnamed pipe does not exist without processes connected to both ends
9. In POSIX message queue, you can configure message size and number of messages
10. In shared memory IPC, the Virtual Memory manager maps the same piece of physical memory to the address space of each sharing process
11. After creating a shared memory segment with `shm_open()` function, the default size of the segment is 0
12. POSIX IPC objects (message queues, shared memory, semaphores) can be found under `/dev/mqueue` and `/dev/shm` directories
13. You can set/change the length of the shared memory segment using `ftruncate()` function
14. In POSIX, names for message queue, shared memory and kernel semaphores must start with a "/"
15. `sem_unlink()` function permanently removes a semaphore from the kernel
16. You must use `ftruncate()` before using a shared memory segment
17. You must call `mmap()` before using (i.e., read/write) a shared memory segment
18. UDP protocol deals with retransmitting packets in case they are lost in route
19. TCP protocol is more heavy-weight because it maintains state information about the connection
20. Routing protocols are dynamic: they reconfigure under network topology change or outage automatically
21. Domain Naming System (DNS) can be used for load balancing and faster content delivery
22. The ping command is used to test whether you can make TCP connections with a remote host
23. The `accept()` function needs to be called the same number of times as the number of client-side `connect()` calls to accept all of them
24. HTTP protocol uses TCP underneath
25. Ports [0,1023] are reserved for well-known services (e.g., HTTP, SMTP, DNS, TELNET)
26. The master socket in a TCP server is used only to accept connections, not to run conversations with clients
27. A socket is a pair of IP-address and port number combination in both client and server side
28. For making a socket on the client side, the port number is chosen by the OS randomly from the available pool
29. A socket is like other file descriptors and is added to the Descriptor Table
30. UDP is connectionless while TCP is connection oriented

**FILE SYSTEMS**

31. [20 pts] Assume that a file system has each disk block of size 4KB and each block pointer of 4 bytes. In addition, the each inode in this system has 14 direct pointers, 2 single indirect pointers, 1 double indirect and 1 triple indirect pointer. Ignoring the space for inode, answer the following questions for this file system:
  - (a) What is the maximum possible file size? [6 pts]
  - (b) How much overhead (amount of non-data information) for the maximum file size derived in (a)? [7 pts]
  - (c) How much overhead for a file of size 6GB? [7 pts]

ANSWER



(a)

In the above, the far-left box is the inode that contains pointers (direct or indirect) to data blocks. Now, according to specification, there 14 direct pointers to disk blocks. Since each block is 4KB, these direct pointers lead us to  $14 \times 4\text{KB} = 96\text{KB}$ .

Now, each single indirect (SI) pointer first points to a pointer block, which is full of pointers to disk blocks. A disk block, irrespective of data or pointer, is always 4KB in size. When it contains pointers, it can contain  $4\text{KB}/4 = 1024$  of those. As a result, we get a tree where the root node is indirect block and then the tree branches out to 1024 leaf nodes that are data blocks. [Notice that I am using red color for pointer blocks and black for data blocks. We will use this drawing for parts (b) and (c)]. That means, each SI pointer can lead us to:  $1024 \times 4\text{KB} = 4\text{MB}$  worth of data. Since there are 2 SI pointers, the capacity at this level is = 8MB.

The same rationale applies to the DI pointers, except that now we have a deeper tree with 2 levels of internal nodes and 1 level of leaf nodes. Using our knowledge of tree, we can calculate that the leaf level now has  $1024^2$  nodes giving us:  $1024^2 * 4KB = 4GB$  capacity.

The TI indirect level, in the same way, gives us:  $1024^3 * 4KB = 4TB$  data.

Therefore, the max file size is just all the above added together:  $96KB + 8MB + 4GB + 4TB$ . [In exam, for similar questions, just stop here – no need to calculate any further]

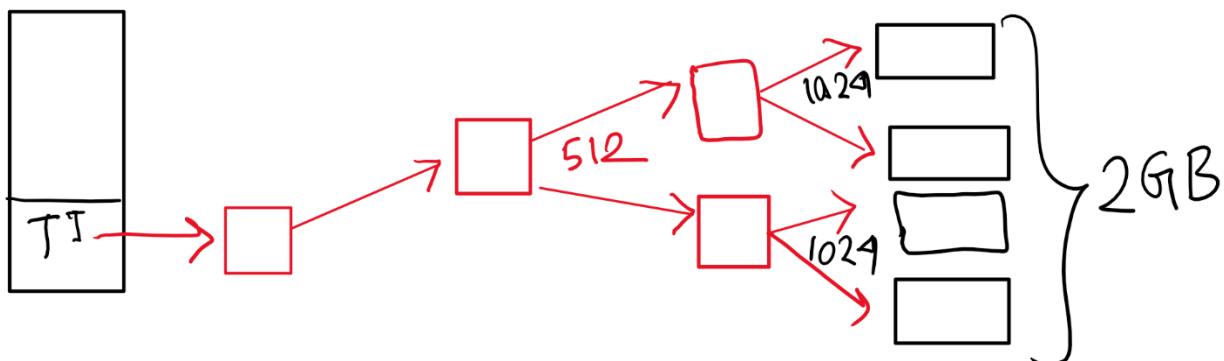
(b)

Now, to represent a max size file, you need to just count the red blocks in the above picture, because they are the ones that do not hold any data. That is my definition of overhead for this type of question. [From that perspective, technically speaking, even the inode itself is an overhead. But we are not considering the inode overhead here, because: (1) I want to keep things simple, (2) an inode does not take an entire disk block.] In other words, you just have to count only the internal nodes for all the pointer trees.

$$\begin{aligned}\text{Total overhead} &= 2 \text{ blocks for 2 SI} + (1 + 1024) \text{ blocks for 1 DI} + (1 + 1024 + 1024^2) \text{ blocks for 1 TI} \\ &= (2 + 1 + 1024 + 1 + 1024 + 1024^2) * 4KB\end{aligned}$$

(c) The total overhead for a 6GB file: Using up to DI level, we can reach a file capacity of  $(96KB + 8MB + 4GB)$ . That means, about 2GB is yet to be represented using the TI level (note that I am simplifying it a little bit. For exact calculation, you have to subtract the whole underlined amount from 6GB and represent the rest using TI. But we do not want to use the calculator, so it is OK to do such simplification).

Now, to represent the remaining 2GB, you do not need the entire TI level, only part of it. Therefore, the overhead will be less than that of the maximum sized file. In other words, you will only need part of the TI tree and we will only calculate for that part. To get to 2GB, the tree you need looks like the following:



This shows that we are using 1 pointer out of the TI root (the rest 1023 pointers are not needed) and 512 pointers out of the block pointed by that. The remaining 512 pointers are not needed, because we already reach 2GB by fully expanding 1024 ways the last level (i.e., before leaf level) internal nodes. Also, even the partially used pointer blocks are fully overhead (i.e., non-data) because you cannot keep any data there. This leads to the overhead of:  $(1 + 1 + 512) * 4KB$  worth of disk being “wasted” in the TI

level. You also have to add the overhead from SI and DI levels who total to  $(2 + 1 + 1024) * 4KB$ . Putting all together, we get an overhead of  $= (2 + 1 + 1024 + 1 + 1 + 512) * 4KB$ , whatever that number is.

## SIGNALS

32. [10 pts] The following code will create a Zombie child process because the child process is terminated and the parent process is busy in a loop without calling `wait()` function. Now, modify this program by handling `SIGCHLD` signal so that no Zombie process is created. The parent process cannot call `wait()` directly in the `main()`. However, calling `wait()` from inside the signal handler is fine. The main still must go to the infinite while loop. You can add helper functions.

```
int main(){
    if (fork() == 0) // child process
        exit(0);
    else // parent process
        while (true);
}
```

## ANSWER:

```
int handler(int s){
    wait (NULL);
}

int main(){
    signal (SIGCHLD, handler); // install the signal handler here
    if (fork() == 0) // child process
        exit(0);
    else { // parent process
        // or here: signal (SIGCHLD, handler);
        while (true);
    }
}
```

33. [15 pts] Consider the program below and answer the following questions with proper explanation.

- What is the output? How much time does the program take to run? [10 points]
- What is the output with line 5 commented? How much time will it take now? [5 points]

```
1 void signal_handler (int signo){
2     printf ("Got SIGUSR1\n");
3 }
4 int main () {
5     signal (SIGUSR1, signal_handler); //comment out for b)
6     int pid = fork ();
7     if (pid == 0){ // child process
8         for (int i=0; i<5; i++){
9             kill(getppid(), SIGUSR1);
10            sleep (1);
11        }
12    }else{ // parent process
13        wait(0);
14    }
15 }
```

Answer:

Here, we have a child process that is supposed to runs for ~5 seconds (because of the sleep(1) 5 times from the loop, everything else takes minimal time) and the parent process, if alive, will wait for the child process, because of the wait(0) statement in line 13. Now, whether the parent process will live or not, will depend on the its ability to handle the SIGUSR1. Note that the child always runs for 5 seconds and that does not change. It is the parent process's lifetime that we are trying to determine.

- (a) 5 seconds, because you have a signal handler. Output is the prompt 5 times:

```
Got SIGUSR1
Got SIGUSR1
Got SIGUSR1
Got SIGUSR1
Got SIGUSR1
```

- (b) ~0 seconds. Because there is no handler for the SIGUSR1, which kills the parent process in the first instance.

34. [15 pts] Write a wrapper class KernelSemaphore on top of POSIX kernel semaphore. See sem\_overview(7) in man pages or linux.die.net to learn about kernel semaphores. Test your KernelSemaphore class by by setting the initial value to 0. Then write 2 programs – one waits for the semaphore and the other one releases (i.e., V()) it. The header for the KernelSemaphore and the 2 programs in questions are provided in the below. You should make sure that the consumer program can only print out its prompt after the producer program has released the semaphore. [Answer is provided in the below, you only need to provide a full definition of kernel semaphore class]

```
class KernelSemaphore{
    string name;
    sem_t* sema;
public:
    KernelSemaphore (string _name, int _init_value){
        name = _name; //retain name because the destructor will need it
        sema = sem_open (name.c_str(), O_CREAT, 0644, _init_value);
    }
    void P(){sem_wait (sema);}
    void V(){sem_post (sema);}
    ~KernelSemaphore (){
        sem_close (sema); //removes the process's handle to the sema, but
        does not remove from the kernel, which is done in the next step
        sem_unlink(name.c_str()); //note you refer to the name here. That is
        why the constructor needs to retain the name of the kernel semaphore
    }
};

// producer.cpp (Run this first in a terminal)
int main (){
    cout << "This program will create the semaphore, initialize it to 0, ";
    cout << "then produce some data and finally V() the semaphore" << endl;
    KernelSemaphore ks ("/my_kernel_sema", 0);
    sleep (rand () % 10); // sleep a random amount of seconds
    ks.V();
}

// consumer.cpp (Run this second in another terminal)
int main (){
}
```

```
KernelSemaphore ks ("/my_kernel_sema", 0);
ks.P();
cout << "I can tell the producer is done" << endl;
}
```

35. [10 pts] Assume a very old computer system from a company that used a very old legacy device whose path is /dev/legacy/specialdevice and it is about to be decommissioned. However, there are several important pieces of software who use this legacy device to log their output and you cannot change those. That means, the path /dev/legacy/specialdevice must continue to exist although the underlying physical device must be replaced. Now, what can you do to make sure all legacy tools and software continues running w/o problem without putting a new physical device in the above path? Note: you may forward all traffic to the legacy device to let's say /sys/logfile path.

Answer: Make /dev/legacy/specialdevice a symbolic link of the actual path /sys/logfile