

CSI 62
Operating Systems and
Systems Programming
Lecture 9

Sockets, Networking (Con't)
Scheduling

February 20th, 2020

Prof. John Kubiawicz

<http://cs162.eecs.Berkeley.edu>

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiawicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.

Recall: POSIX I/O: Everything (looks like) a “File”

- Identical interface for:
 - Devices (terminals, printers, etc.)
 - Regular files on disk
 - Networking (sockets)
 - Local interprocess communication (pipes, sockets)
- Based on `open()`, `read()`, `write()`, and `close()`
- Allows simple composition of programs
 - » `find | grep | wc ...`
- **HOWEVER: Not every thing actually IS a file!**
 - Pipes are only buffered in memory!
 - Network sockets only buffered in memory/network!

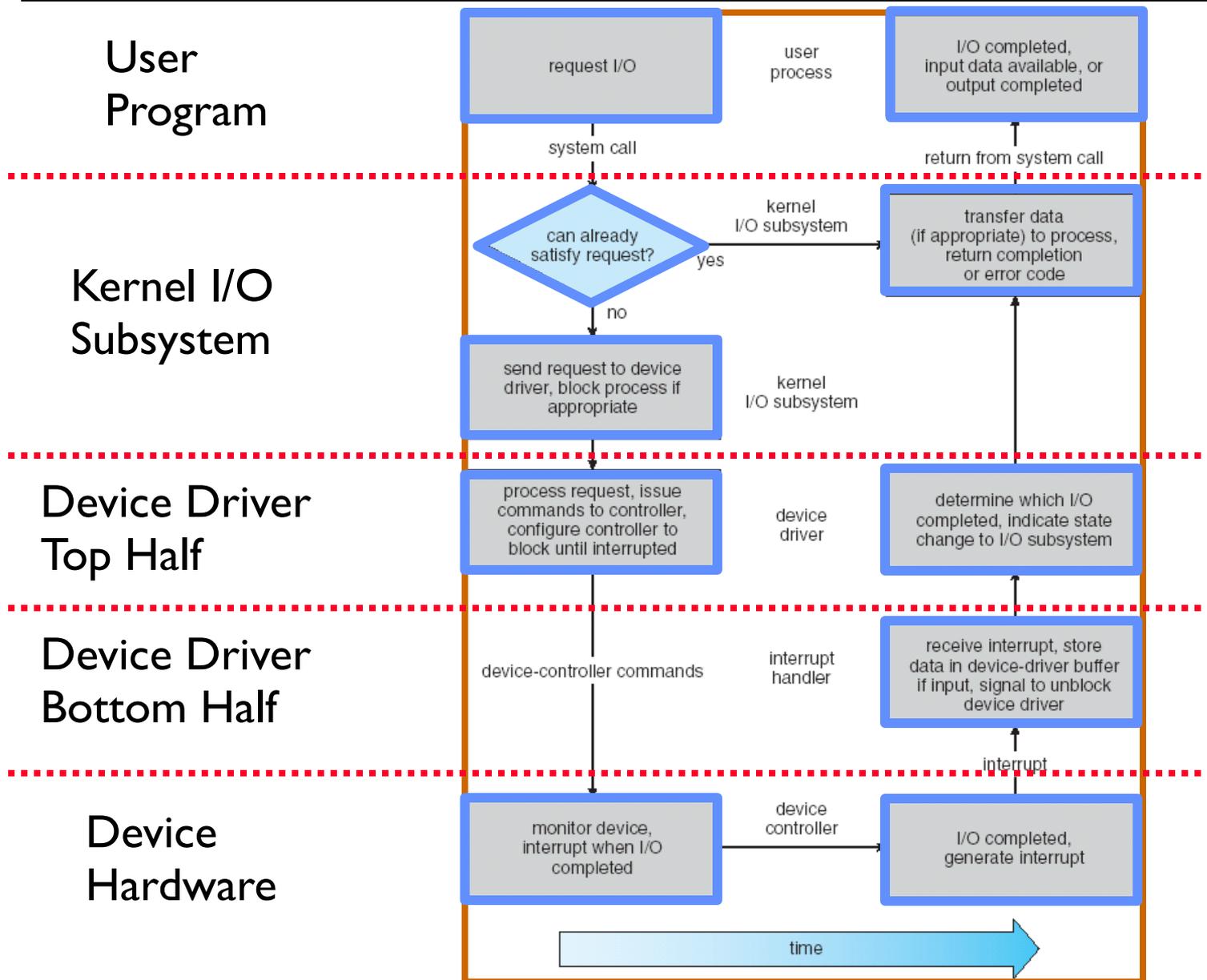
Recall: POSIX I/O Design Patterns

- Open before use
 - Access control check, setup happens here
- Byte-oriented
 - Least common denominator
 - OS responsible for hiding the fact that real devices may not work this way (e.g. hard drive stores data in blocks)
- Explicit close
- Reads are buffered
 - Part of making everything byte-oriented
 - Process is blocked while waiting for device
 - Let other processes run while gathering result
- Writes are buffered
 - Complete in background (more later on)
 - Return to user when data is “handed off” to kernel
- Errors relayed to user in a variety of ways!
 - Make sure to check them!

Recall: Device Drivers

- **Device Driver:** Device-specific code in the kernel that interacts directly with the device hardware
 - Supports a standard, internal interface
 - Same kernel I/O system can interact easily with different device drivers
 - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers typically divided into two pieces:
 - Top half: accessed in call path from system calls
 - » implements a set of **standard, cross-device calls** like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
 - » This is the kernel's interface to the device driver
 - » Top half will *start* I/O to device, may put thread to sleep until finished
 - Bottom half: run as interrupt routine
 - » Gets input or transfers next block of output
 - » May wake sleeping threads if I/O now complete

Recall: Life Cycle of An I/O Request



Communication between processes

- Can we view files as communication channels?

```
write(wfd, wbuf, wlen);
```

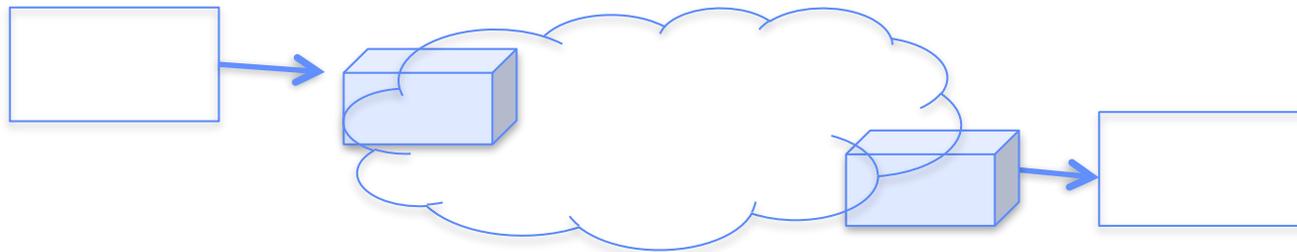


```
n = read(rfd, rbuf, rmax);
```

- Producer and Consumer of a file may be distinct processes
 - May be separated in time (or not)
- However, what if data written once and consumed once?
 - Don't we want something more like a queue?
 - Can still look like File I/O!

Communication Across the world looks like file IO

```
write(wfd, wbuf, wlen);
```



```
n = read(rfd, rbuf, rmax);
```

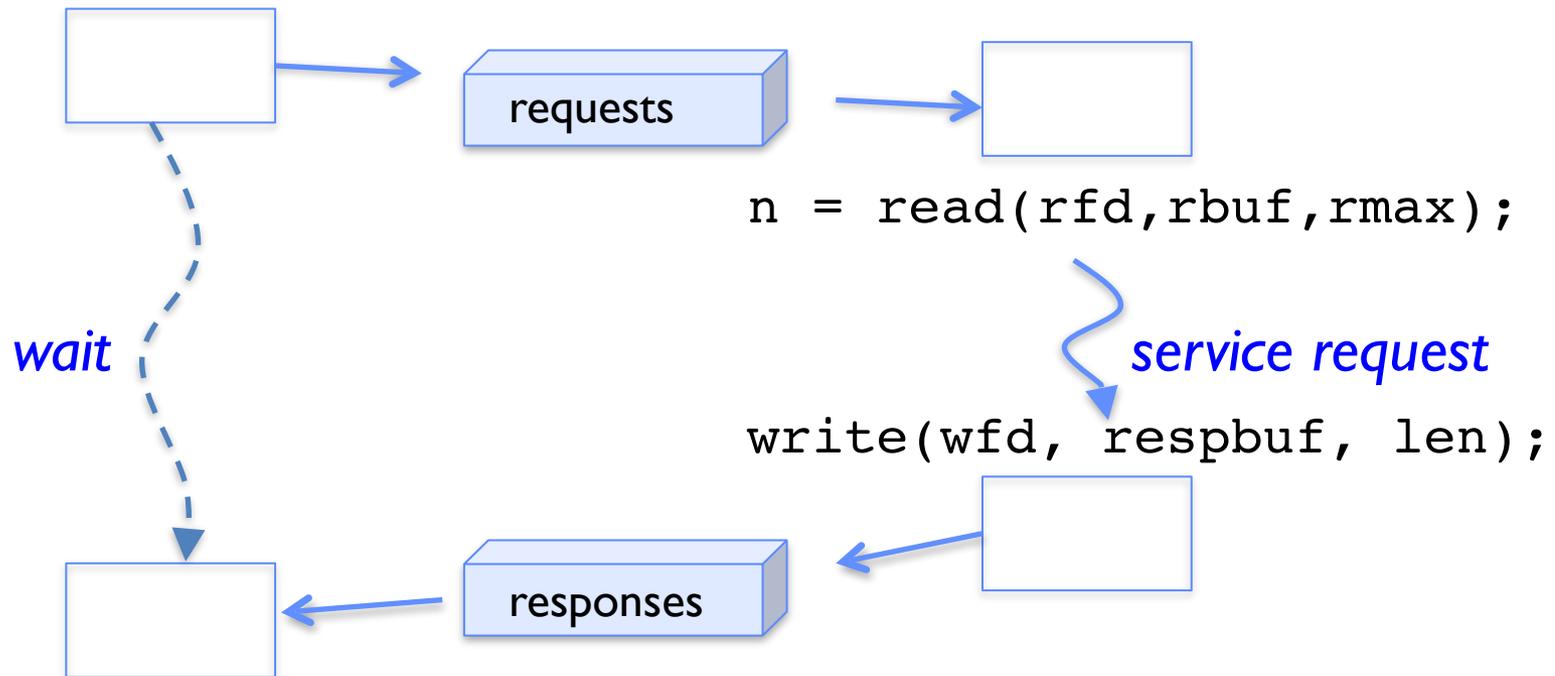
- Connected queues over the Internet
 - But what's the analog of open?
 - What is the namespace?
 - How are they connected in time?

Request Response Protocol

Client (issues requests)

Server (performs operations)

```
write(rqfd, rqbuf, buflen);
```



```
n = read(rfd, rbuf, rmax);
```

```
write(wfd, resbuf, len);
```

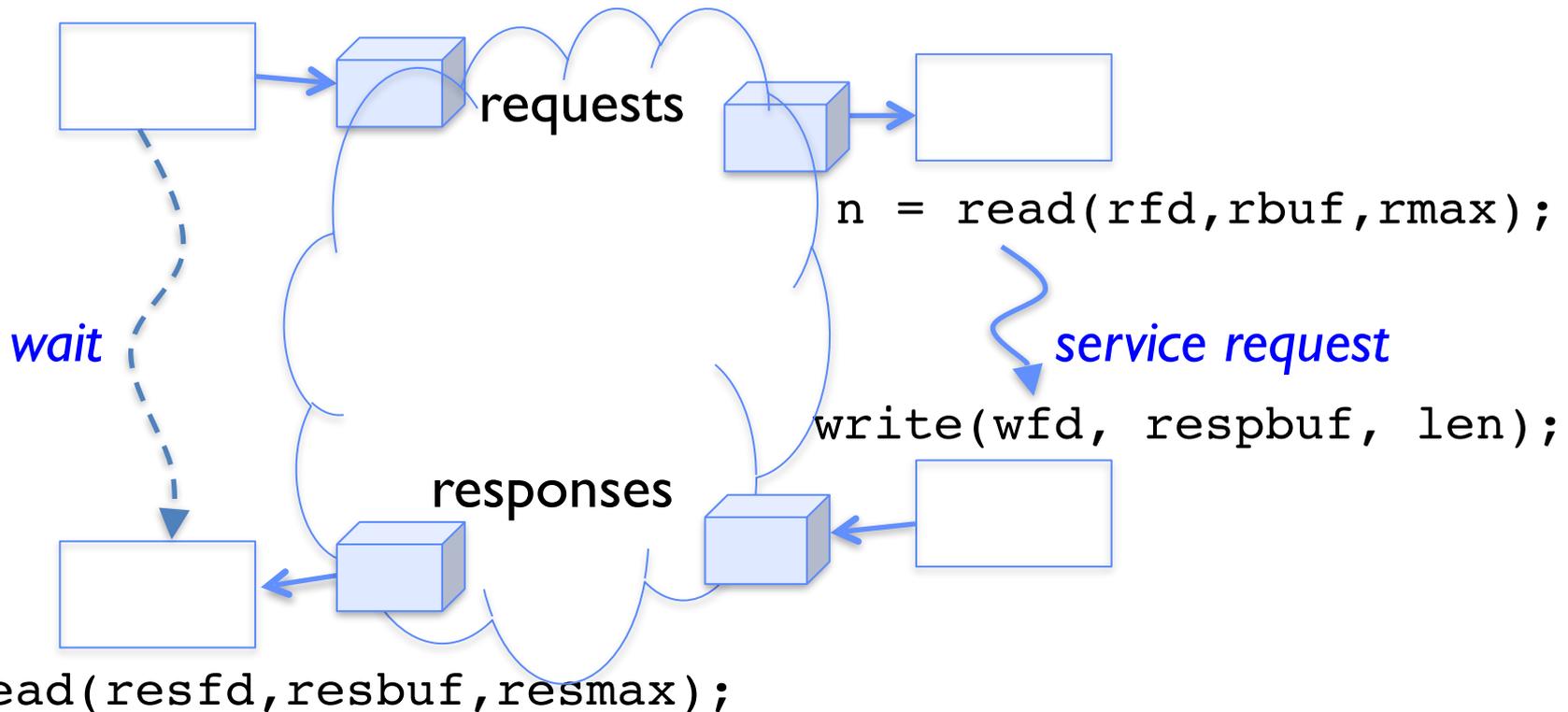
```
n = read(resfd, resbuf, resmax);
```

Request Response Protocol

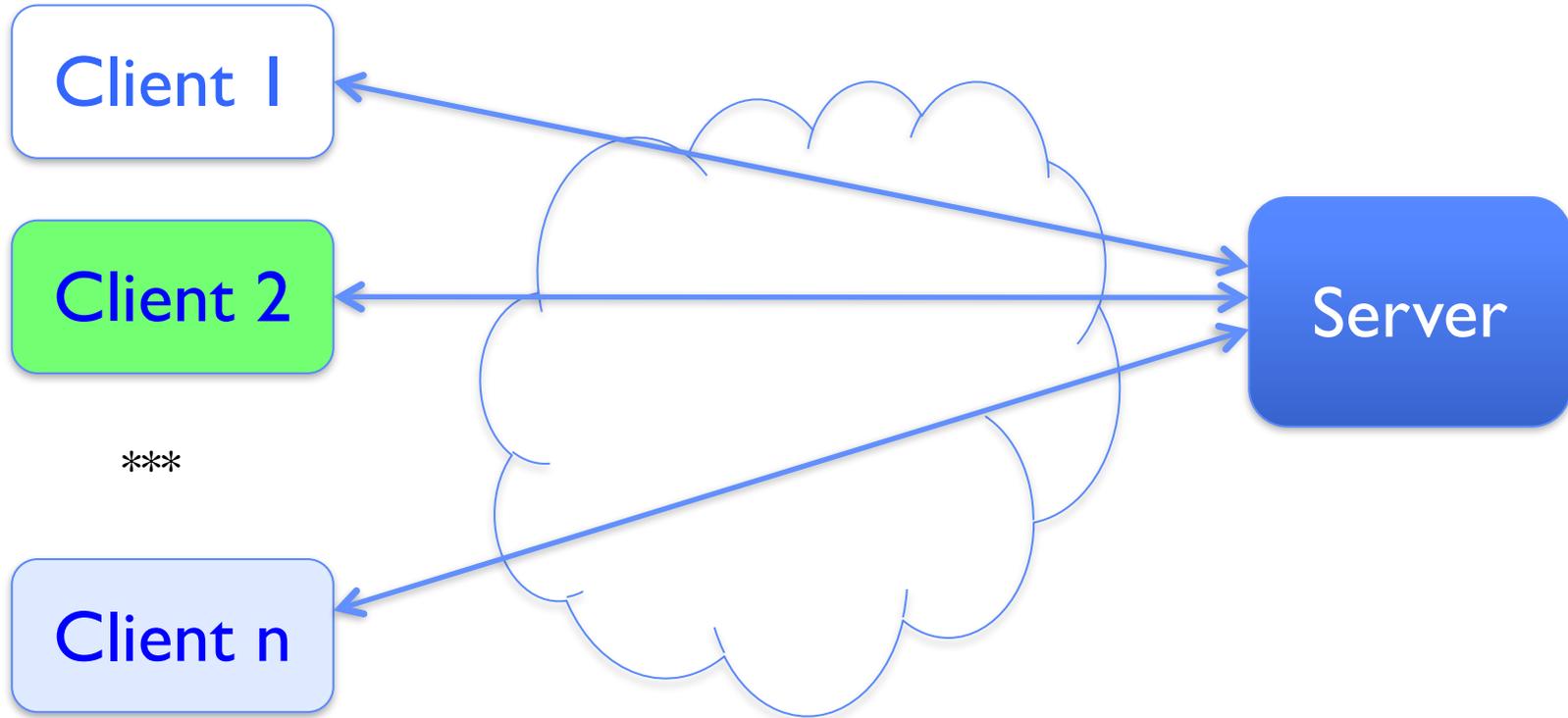
Client (issues requests)

Server (performs operations)

```
write(rqfd, rqbuf, buflen);
```



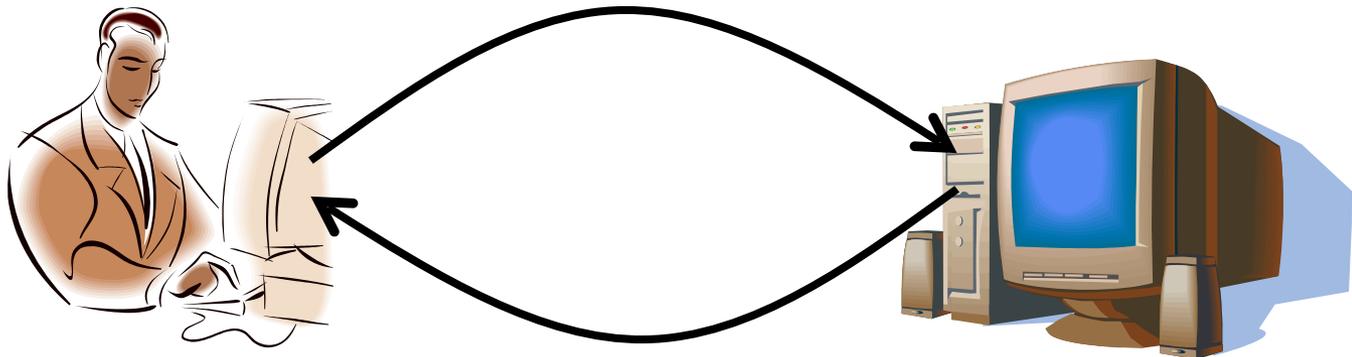
Client-Server Models



- File servers, web, FTP, Databases, ...
- Many clients accessing a common server

Client-Server Communication

- Client “sometimes on”
 - Initiates a request to the server when interested
 - E.g., Web browser on your laptop or cell phone
 - Doesn’t communicate directly with other clients
 - Needs to know the server’s address
- Server is “always on”
 - Services requests from many client hosts
 - E.g., Web server for the *www.cnn.com* Web site
 - Doesn’t initiate contact with the clients
 - Needs a fixed, well-known address



Sockets

- **Socket**: an abstraction of a network I/O queue
 - Mechanism for inter-process communication
 - Embodies one side of a communication channel
 - » Same interface regardless of location of other end
 - » Could be local machine (called “UNIX socket”) or remote machine (called “network socket”)
 - First introduced in 4.2 BSD UNIX: big innovation at time
 - » Now most operating systems provide some notion of socket
- Data transfer like files
 - Read / Write against a descriptor
- Over ANY kind of network
 - Local to a machine
 - Over the internet (TCP/IP, UDP/IP)
 - OSI, Appletalk, SNA, IPX, SIP, NS, ...

Silly Echo Server – running example

Client (issues requests)

Server (performs operations)



```
gets (fd, sndbuf, ...);
```

```
write (fd, buf, len);
```

wait

```
n = read (fd, rcvbuf, );
```

print



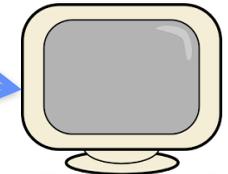
requests

responses

```
n = read (fd, buf, );
```

print

```
write (fd, buf, );
```



Echo client-server example

```
void client(int sockfd) {
    int n;
    char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
    getreq(sndbuf, MAXIN);          /* prompt */
    while (strlen(sndbuf) > 0) {
        write(sockfd, sndbuf, strlen(sndbuf)); /* send */
        memset(rcvbuf, 0, MAXOUT);          /* clear */
        n=read(sockfd, rcvbuf, MAXOUT-1);   /* receive */
        write(STDOUT_FILENO, rcvbuf, n);    /* echo */
        getreq(sndbuf, MAXIN);             /* prompt */
    }
}
```

```
void server(int consockfd) {
    char reqbuf[MAXREQ];
    int n;
    while (1) {
        memset(reqbuf, 0, MAXREQ);
        n = read(consockfd, reqbuf, MAXREQ-1); /* Recv */
        if (n <= 0) return;
        n = write(STDOUT_FILENO, reqbuf, strlen(reqbuf));
        n = write(consockfd, reqbuf, strlen(reqbuf)); /*
echo*/
    }
}
```

What assumptions are we making?

- Reliable
 - Write to a file => Read it back. Nothing is lost.
 - Write to a (TCP) socket => Read from the other side, same.
 - Like pipes
- In order (sequential stream)
 - Write X then write Y => read gets X then read gets Y
- When ready?
 - File read gets whatever is there at the time. Assumes writing already took place.
 - Like pipes!

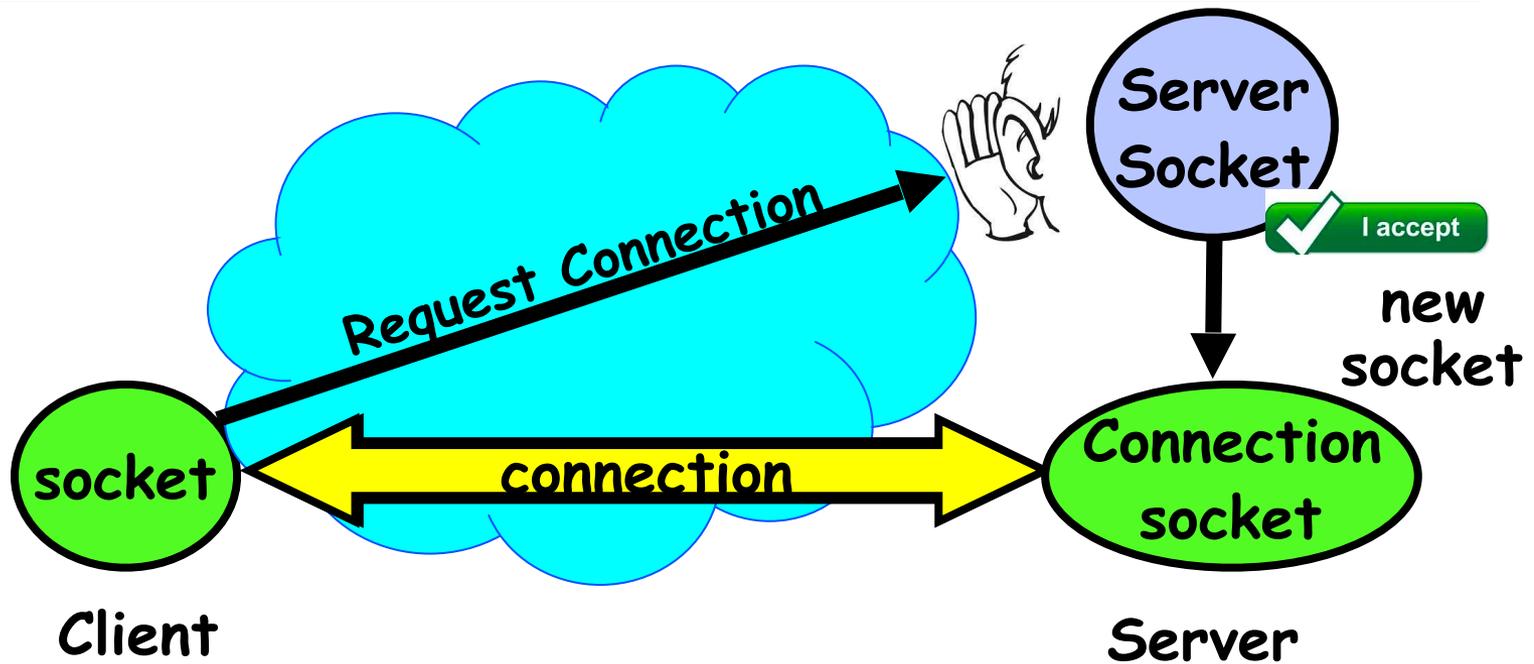
Socket creation and connection

- File systems provide a collection of permanent objects in structured name space
 - Processes open, read/write/close them
 - Files exist independent of the processes
- Sockets provide a means for processes to communicate (transfer data) to other processes.
- Creation and connection is more complex
- Form 2-way pipes between processes
 - Possibly worlds away
- How do we name them?
- How do these completely independent programs know that the other wants to “talk” to them?

Namespaces for communication over IP

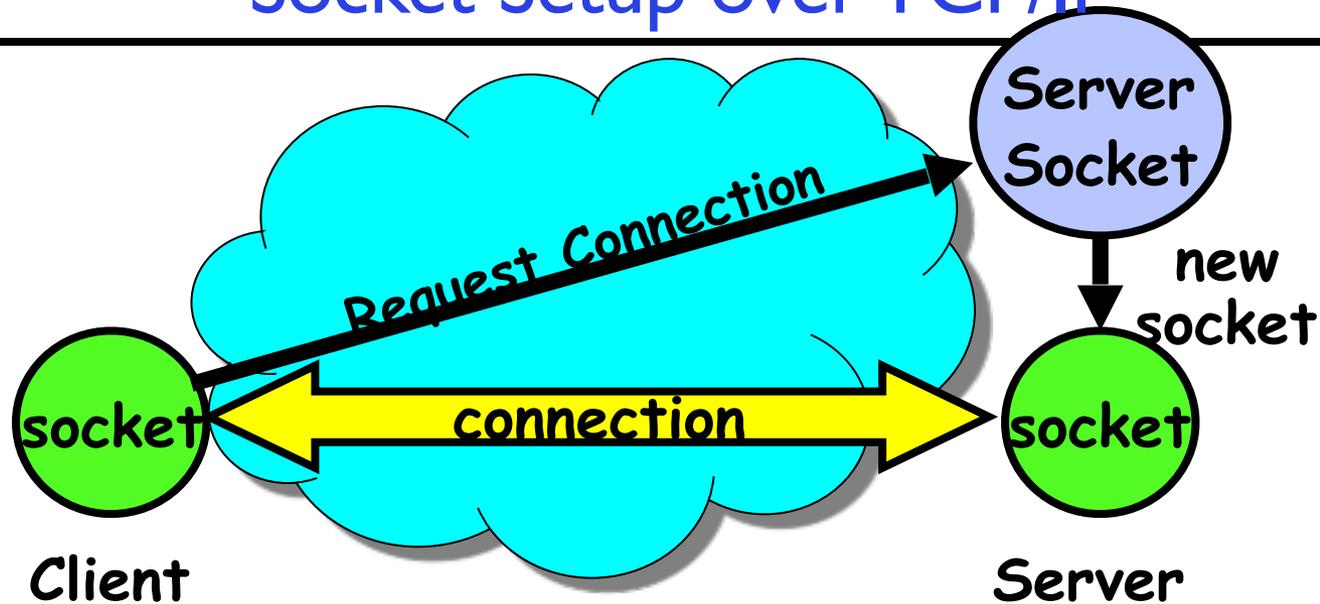
- Hostname
 - www.eecs.berkeley.edu
- IP address
 - 128.32.244.172 (ipv6?)
- Port Number
 - 0-1023 are “well known” or “system” ports
 - » Superuser privileges to bind to one
 - 1024 – 49151 are “registered” ports (registry)
 - » Assigned by IANA for specific services
 - 49152–65535 ($2^{15}+2^{14}$ to $2^{16}-1$) are “dynamic” or “private”
 - » Automatically allocated as “ephemeral Ports”

Socket Setup over TCP/IP



- Special kind of socket: server socket
 - Has file descriptor
 - Can't read or write
- Two operations:
 1. **listen()**: Start allowing clients to connect
 2. **accept()**: Create a *new socket* for a *particular* client connection

Socket Setup over TCP/IP

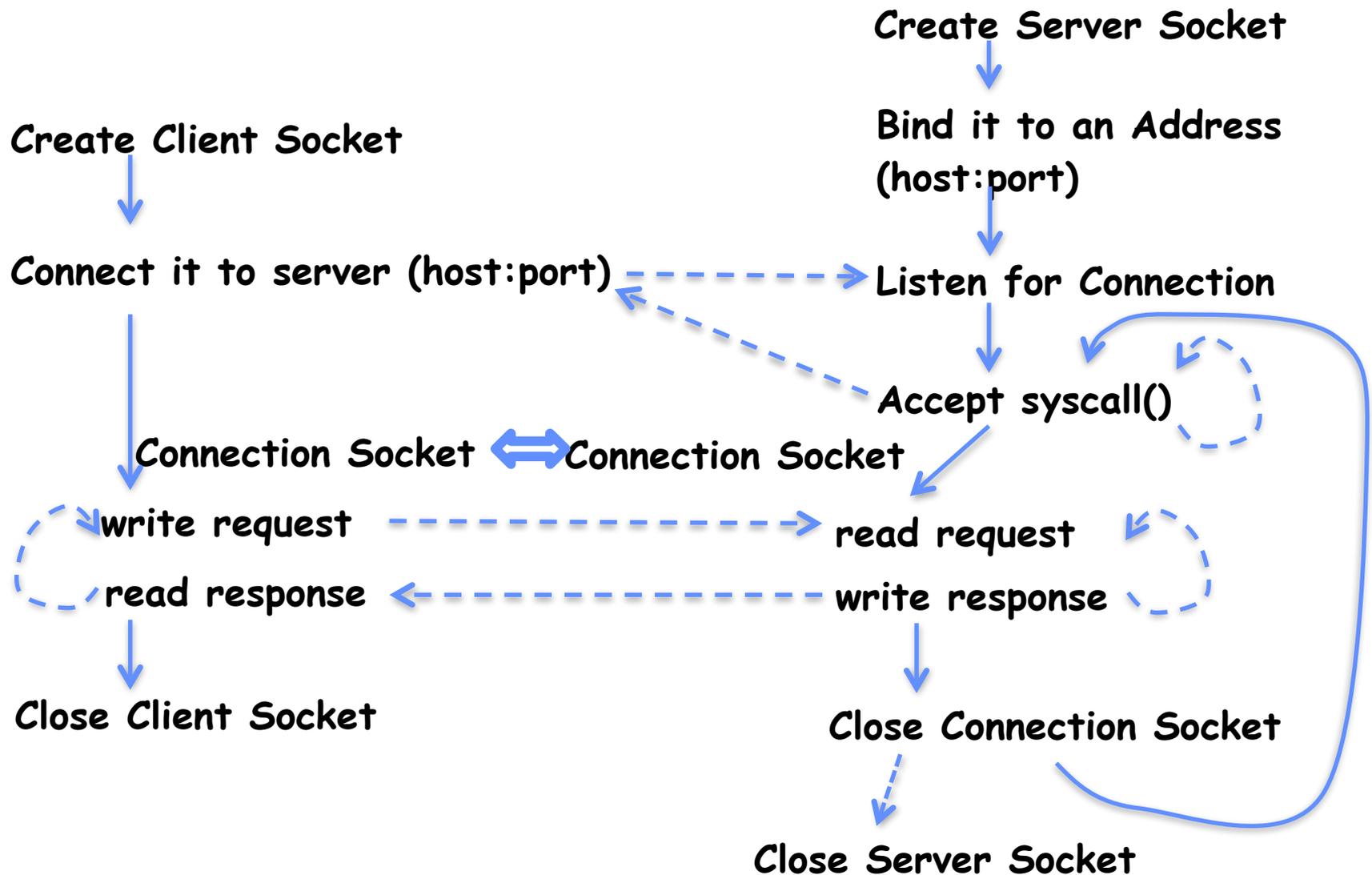


- Server Socket: Listens for new connections
 - Produces new sockets for each unique connection
 - 3-way handshake to establish new connection!
- Things to remember:
 - Connection involves 5 values:
[Client Addr, Client Port, Server Addr, Server Port, Protocol]
 - Often, Client Port “randomly” assigned
 - » Done by OS during client socket setup
 - Server Port often “well known”
 - » 80 (web), 443 (secure web), 25 (sendmail), etc
 - » Well-known ports from 0—1023

Web Server using Sockets (in concept)

Client

Server



Client-Side of Protocol

```
char *host_name, port_name;

// Create a socket
struct addrinfo *server = lookup_host(host_name, port_name);
int sock_fd = socket(server->ai_family, server->ai_socktype,
                    server->ai_protocol);

// Connect to specified host and port
connect(sock_fd, server->ai_addr, server->ai_addrlen);

// Carry out Client-Server protocol
run_client(sock_fd);

/* Clean up on termination */
close(sock_fd);
```

Client: getting the server address (as addrinfo)

```
struct addrinfo *lookup_host(char *host_name, char *port) {
    struct addrinfo *server;
    struct addrinfo hints;

    // Constraints on returned address
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;    // Either IPv4 or IPv6
    hints.ai_socktype = SOCK_STREAM; // Reliable stream (i.e. TCP)

    // Lookup host:port, constrained by hints, return ptr in
server
    int rv = getaddrinfo(host_name, port_name, &hints, &server);
    if (rv != 0) {
        printf("getaddrinfo failed: %s\n", gai_strerror(rv));
        return NULL;
    }
    return server;
}
```

Server Protocol (v1)

```
// Create socket to listen for client connections
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family, server-
                           >ai_socktype, server->ai_protocol);

// Bind socket to specific port
bind(server_socket, server->ai_addr, server->ai_addrlen);

// Start listening for new client connections
listen(server_socket, MAX_QUEUE);

while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);
    serve_client(conn_socket);
    close(conn_socket);
}

close(server_socket);
```

Server: getting server addrinfo – for itself

```
struct addrinfo *setup_address(char *port) {
    struct addrinfo *server;
    struct addrinfo hints;

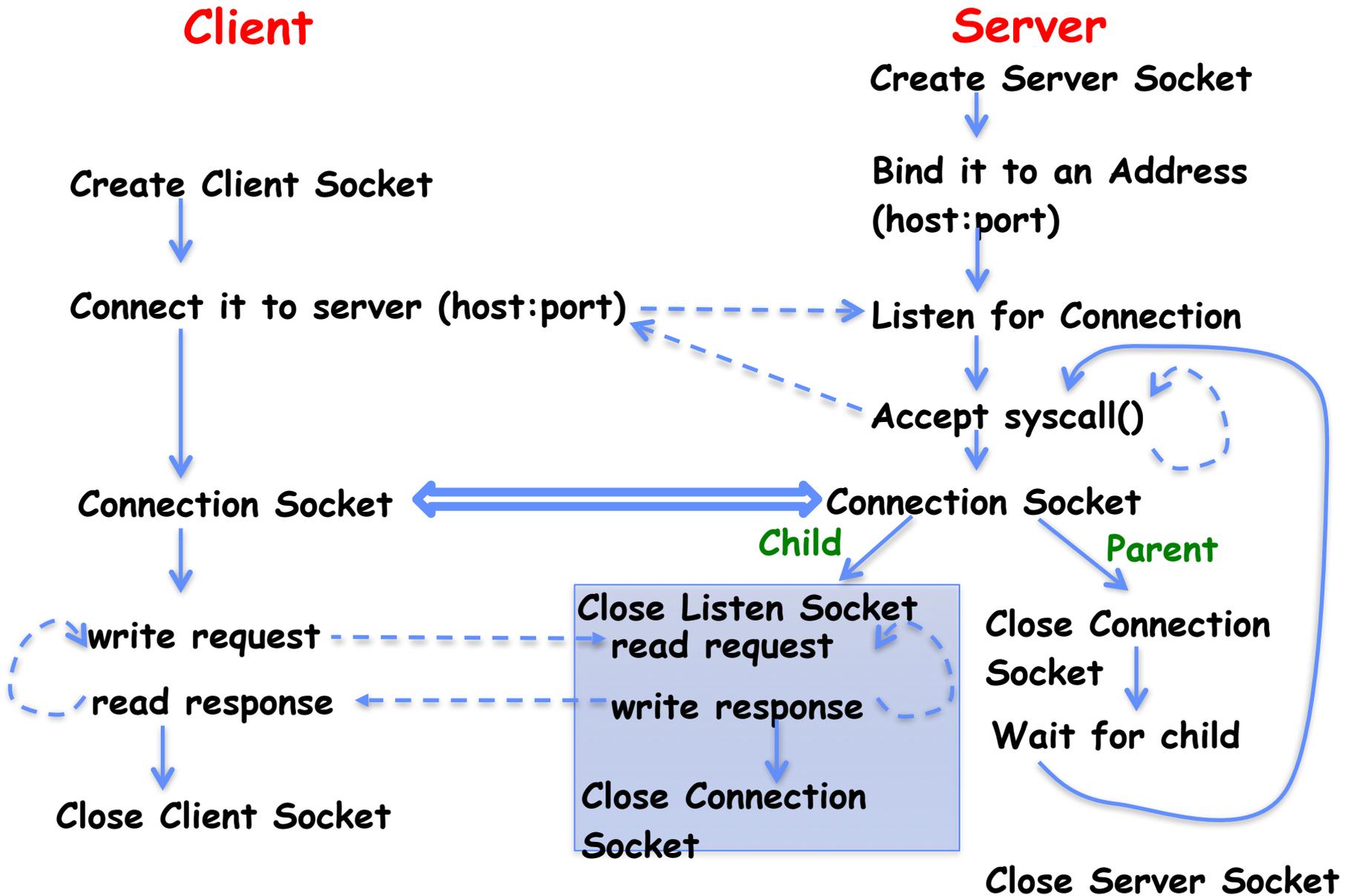
    // Constraints on returned address
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC; // IPv4 or IPv6
    hints.ai_socktype = SOCK_STREAM; // Reliable stream (i.e. TCP)
    hints.ai_flags = AI_PASSIVE; // Address for listening

    // Match any local address:port, constrained by hints, return
ptr
    int rv = getaddrinfo(NULL, port, &hints, &server);
    if (rv != 0) {
        printf("getaddrinfo failed: %s\n", gai_strerror(rv));
        return NULL;
    }
    return server;
}
```

How does the server protect itself?

- Isolate the handling of each connection
- By forking it off as another process

Sockets With Protection



Server Protocol (v2)

```
// Start listening for new client connections
listen(server_socket, MAX_QUEUE);

while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);

    pid_t pid = fork(); // New process for connection
    if (pid == 0) { // Child process
        close(server_socket); // Doesn't need server_socket
        serve_client(conn_socket); // Serve up content to client
        close(conn_socket); // Done with client!
        exit(EXIT_SUCCESS);
    } else { // Parent process
        close(conn_socket); // Don't need client socket
        wait(NULL); // Wait for our (one) child
    }
}
close(server_socket);
```

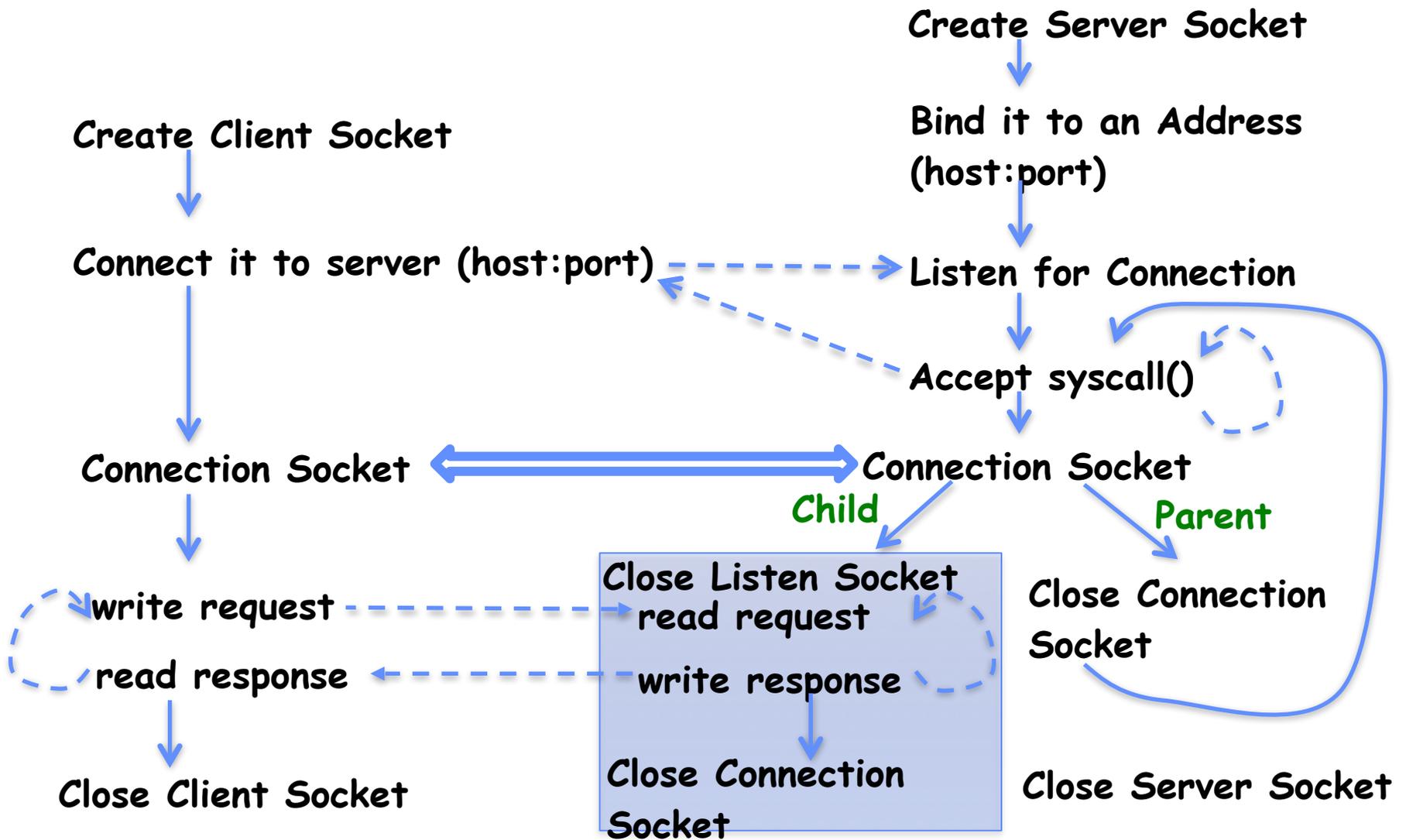
Concurrent Server

- Listen will queue requests
- Buffering present elsewhere
- But server waits for each connection to terminate before initiating the next

Sockets With Protection and Parallelism

Client

Server

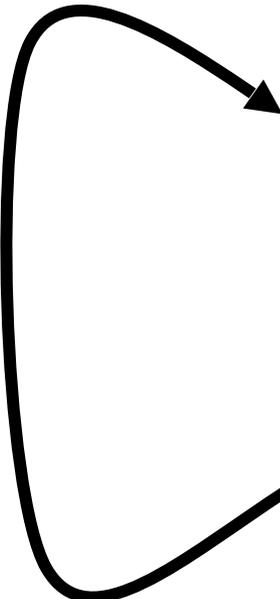


Server Protocol (v3)

```
// Start listening for new client connections
listen(server_socket, MAX_QUEUE);
signal(SIGCHLD, SIG_IGN); // Prevent zombie children
while (1) {
    // Accept a new client connection, obtaining a new socket
    int conn_socket = accept(server_socket, NULL, NULL);

    pid_t pid = fork(); // New process for connection
    if (pid == 0) { // Child process
        close(server_socket); // Doesn't need server_socket
        serve_client(conn_socket); // Serve up content to client
        close(conn_socket); // Done with client!
        exit(EXIT_SUCCESS);
    } else { // Parent process
        close(conn_socket); // Don't need client socket
        // wait(NULL); // Don't wait (SIGCHLD
        // ignored, above)
    }
}
close(server_socket);
```

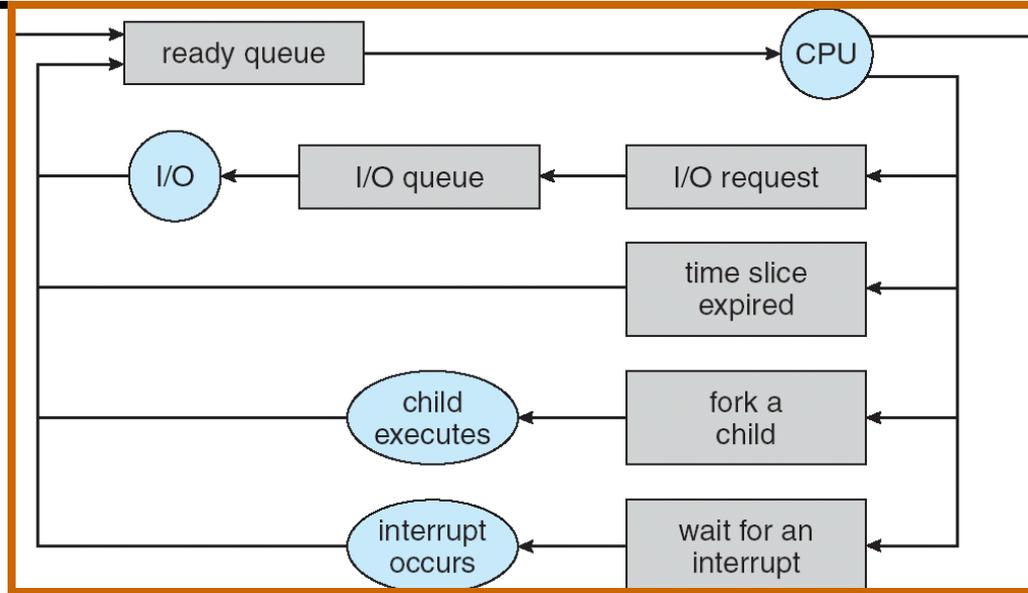
Goal for Today



```
if ( readyThreads(TCBs) ) {
    nextTCB = selectThread(TCBs);
    run( nextTCB );
} else {
    run_idle_thread();
}
```

- Discussion of Scheduling:
 - Which thread should run on the CPU next?
- Scheduling goals, policies
- Look at a number of different schedulers

Recall: Scheduling



- Question: How is the OS to decide which of several tasks to take off a queue?
- **Scheduling**: deciding which threads are given access to resources from moment to moment
 - Often, we think in terms of CPU time, but could also think about access to resources like network BW or disk access

Scheduling: All About Queues

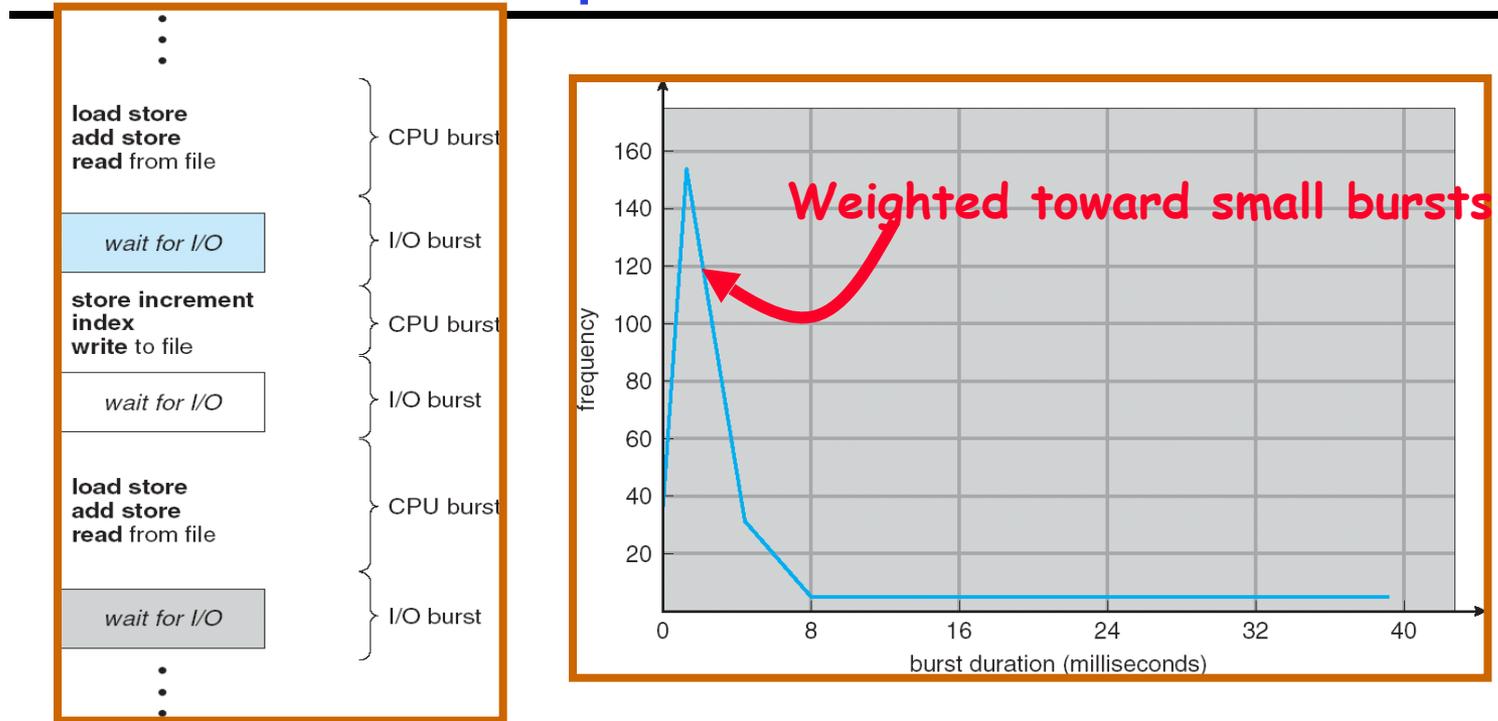


Scheduling Assumptions

- CPU scheduling big area of research in early 70's
- Many implicit assumptions for CPU scheduling:
 - One program per user
 - One thread per program
 - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
 - For instance: is “fair” about fairness among users or programs?
 - » If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system



Assumption: CPU Bursts



- Execution model: programs alternate between bursts of CPU and I/O
 - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
 - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
 - With timeslicing, thread may be forced to give up CPU before finishing current CPU burst

Scheduling Policy Goals/Criteria

- Minimize Response Time
 - Minimize elapsed time to do an operation (or job)
 - Response time is what the user sees:
 - » Time to echo a keystroke in editor
 - » Time to compile a program
 - » Real-time Tasks: Must meet deadlines imposed by World
- Maximize Throughput
 - Maximize operations (or jobs) per second
 - Throughput related to response time, but not identical:
 - » Minimizing response time will lead to more context switching than if you only maximized throughput
 - Two parts to maximizing throughput
 - » Minimize overhead (for example, context-switching)
 - » Efficient use of resources (CPU, disk, memory, etc)
- Fairness
 - Share CPU among users in some equitable way
 - Fairness is not minimizing average response time:
 - » Better *average* response time by making system *less* fair

First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
 - Also “First In, First Out” (FIFO) or “Run until done”
 - » In early systems, FCFS meant one program scheduled until done (including I/O)
 - » Now, means keep CPU until thread blocks



Example:

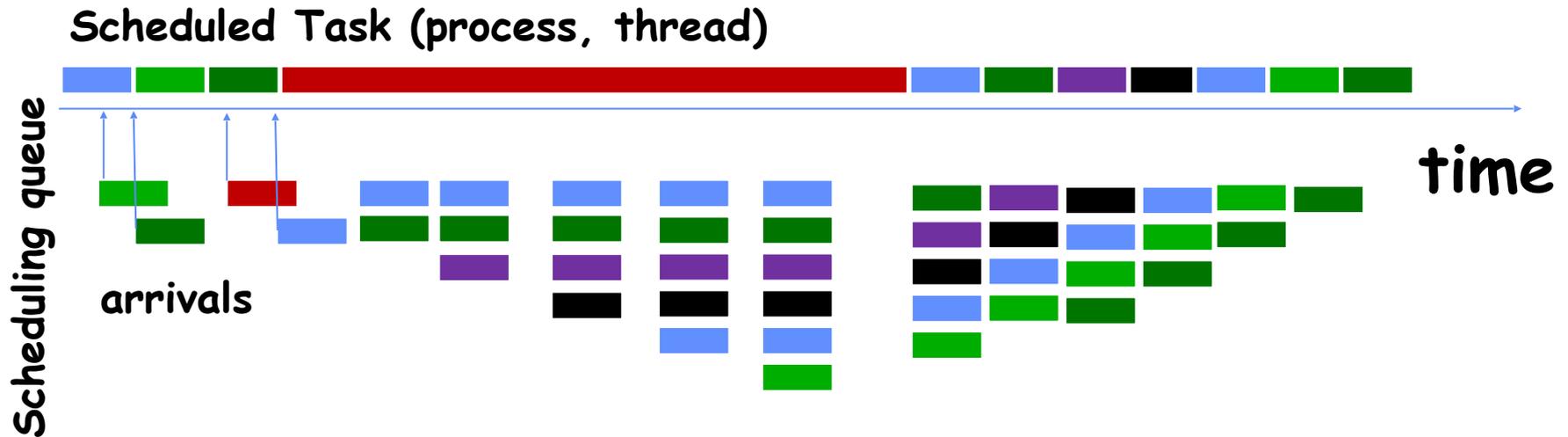
<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

- Suppose processes arrive in the order: P_1, P_2, P_3
The Gantt Chart for the schedule is:



- Waiting time for $P_1 = 0; P_2 = 24; P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
- Average Completion time: $(24 + 27 + 30)/3 = 27$
- *Convoy effect*: short process stuck behind long process

Convoy effect



- With FCFS non-preemptive scheduling, convoys of small tasks tend to build up when a large one is running.

FCFS Scheduling (Cont.)

- Example continued:

- Suppose that processes arrive in order: P₂ , P₃ , P₁
Now, the Gantt chart for the schedule is:



- Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
 - Average waiting time: $(6 + 0 + 3)/3 = 3$
 - Average Completion time: $(3 + 6 + 30)/3 = 13$
- In second case:
 - Average waiting time is much better (before it was 17)
 - Average completion time is better (before it was 27)
 - FIFO Pros and Cons:
 - Simple (+)
 - Short jobs get stuck behind long ones (-)
 - » Getting milk, always stuck behind cart full of items!

Round Robin (RR) Scheduling

- FCFS Scheme: Potentially bad for short jobs!
 - Depends on submit order
 - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand...
- Round Robin Scheme: **Preemption!**
 - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
 - After quantum expires, the process is preempted and added to the end of the ready queue.
 - n processes in ready queue and time quantum is $q \Rightarrow$
 - » Each process gets $1/n$ of the CPU time
 - » In chunks of at most q time units
 - » **No process waits more than $(n-1)q$ time units**



RR Scheduling (Cont.)

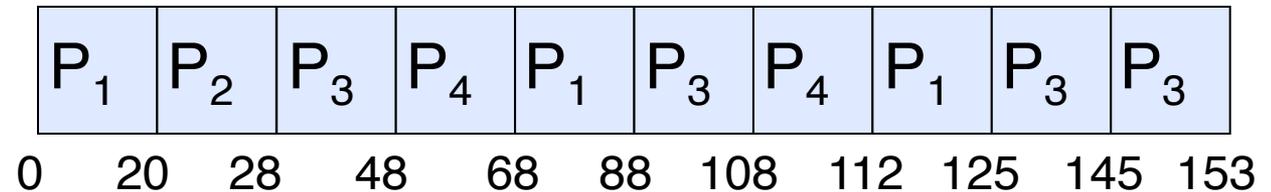
- Performance
 - q large \Rightarrow FCFS
 - q small \Rightarrow Interleaved (really small \Rightarrow hyperthreading?)
 - q must be large with respect to context switch, otherwise overhead is too high (all overhead)

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

– The Gantt chart is:



– Waiting time for

$$P_1 = (68 - 20) + (112 - 88) = 72$$

$$P_2 = (20 - 0) = 20$$

$$P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$$

$$P_4 = (48 - 0) + (108 - 68) = 88$$

– Average waiting time = $(72 + 20 + 85 + 88) / 4 = 66\frac{1}{4}$

– Average completion time = $(125 + 28 + 153 + 112) / 4 = 104\frac{1}{2}$

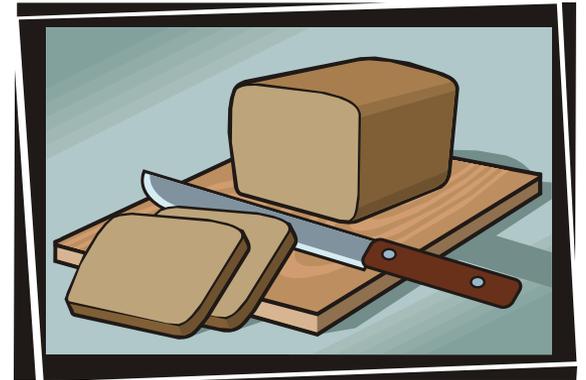
- Thus, Round-Robin Pros and Cons:

– Better for short jobs, Fair (+)

– Context-switching time adds up for long jobs (-)

Round-Robin Discussion

- How do you choose time slice?
 - What if too big?
 - » Response time suffers
 - What if infinite (∞)?
 - » Get back FIFO
 - What if time slice too small?
 - » Throughput suffers!
- Actual choices of timeslice:
 - Initially, UNIX timeslice one second:
 - » Worked ok when UNIX was used by one or two people.
 - » What if three compilations going on? 3 seconds to echo each keystroke!
 - Need to balance short-job performance and long-job throughput:
 - » Typical time slice today is between 10ms – 100ms
 - » Typical context-switching overhead is 0.1ms – 1ms
 - » Roughly 1% overhead due to context-switching



Comparisons between FCFS and Round Robin

- Assuming zero-cost context-switching time, is RR always better than FCFS?

- Simple example:

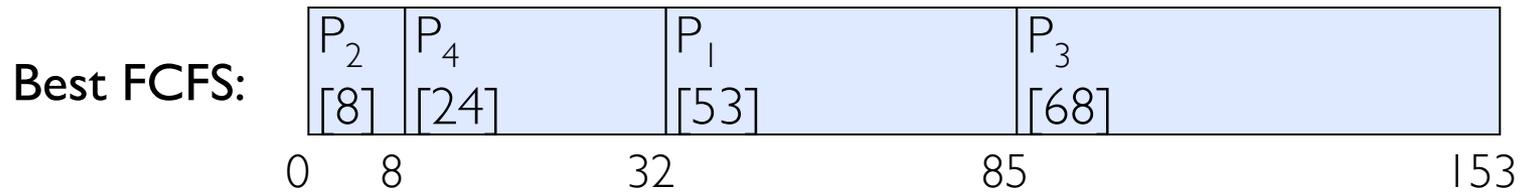
10 jobs, each take 100s of CPU time
RR scheduler quantum of 1s
All jobs start at the same time

- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
...
9	900	999
10	1000	1000

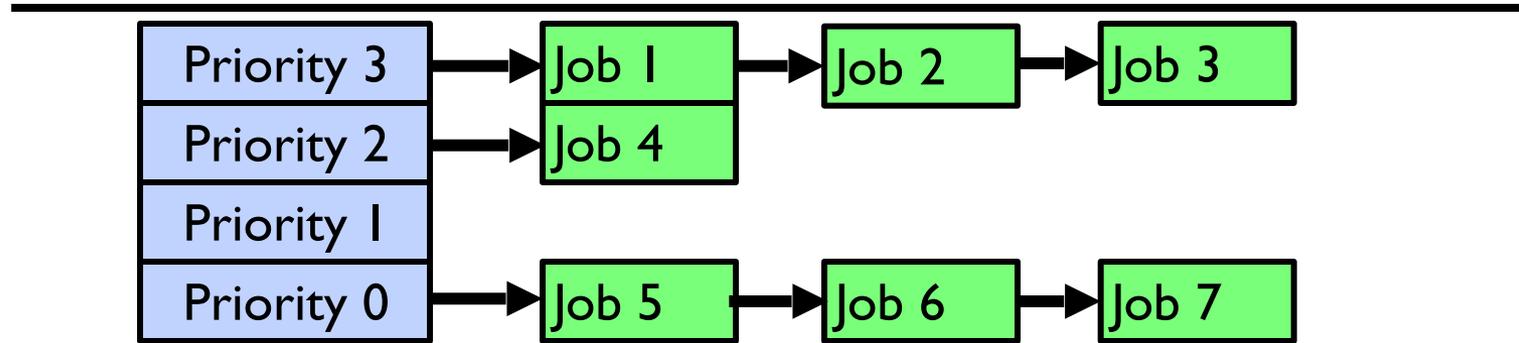
- Both RR and FCFS finish at the same time
- Average response time is much worse under RR!
 - » Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
 - Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quantum



	Quantum	P_1	P_2	P_3	P_4	Average
Wait Time	Best FCFS	32	0	85	8	$31\frac{1}{4}$
	Q = 1	84	22	85	57	62
	Q = 5	82	20	85	58	$61\frac{1}{4}$
	Q = 8	80	8	85	56	$57\frac{1}{4}$
	Q = 10	82	10	85	68	$61\frac{1}{4}$
	Q = 20	72	20	85	88	$66\frac{1}{4}$
	Worst FCFS	68	145	0	121	$83\frac{1}{2}$
Completion Time	Best FCFS	85	8	153	32	$69\frac{1}{2}$
	Q = 1	137	30	153	81	$100\frac{1}{2}$
	Q = 5	135	28	153	82	$99\frac{1}{2}$
	Q = 8	133	16	153	80	$95\frac{1}{2}$
	Q = 10	135	18	153	92	$99\frac{1}{2}$
	Q = 20	125	28	153	112	$104\frac{1}{2}$
	Worst FCFS	121	153	68	145	$121\frac{3}{4}$

Handling Differences in Importance: Strict Priority Scheduling



- Execution Plan
 - Always execute highest-priority runnable jobs to completion
 - Each queue can be processed in RR with some time-quantum
- Problems:
 - Starvation:
 - » Lower priority jobs don't get to run because higher priority jobs
 - Deadlock: Priority Inversion
 - » Not strictly a problem with priority scheduling, but happens when low priority task has lock needed by high-priority task
 - » Usually involves third, intermediate priority task that keeps running even though high-priority task should be running
- How to fix problems?
 - Dynamic priorities – adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc...

Scheduling Fairness

- What about fairness?
 - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
 - » long running jobs may never get CPU
 - » Urban legend: In Multics, shut down machine, found 10-year-old job \Rightarrow Ok, probably not...
 - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
 - Tradeoff: fairness gained by hurting avg response time!

Scheduling Fairness

- How to implement fairness?
 - Could give each queue some fraction of the CPU
 - » What if one long-running job and 100 short-running ones?
 - » Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines
 - Could increase priority of jobs that don't get service
 - » What is done in some variants of UNIX
 - » This is ad hoc—what rate should you increase priorities?
 - » And, as system gets overloaded, no job gets CPU time, so everyone increases in priority⇒Interactive jobs suffer

Lottery Scheduling



- Yet another alternative: Lottery Scheduling
 - Give each job some number of lottery tickets
 - On each time slice, randomly pick a winning ticket
 - On average, CPU time is proportional to number of tickets given to each job
- How to assign tickets?
 - To approximate SRTF, short running jobs get more, long running jobs get fewer
 - To avoid starvation, every job gets at least one ticket (everyone makes progress)
- Advantage over strict priority scheduling: behaves gracefully as load changes
 - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example (Cont.)

- Lottery Scheduling Example
 - Assume short jobs get 10 tickets, long jobs get 1 ticket

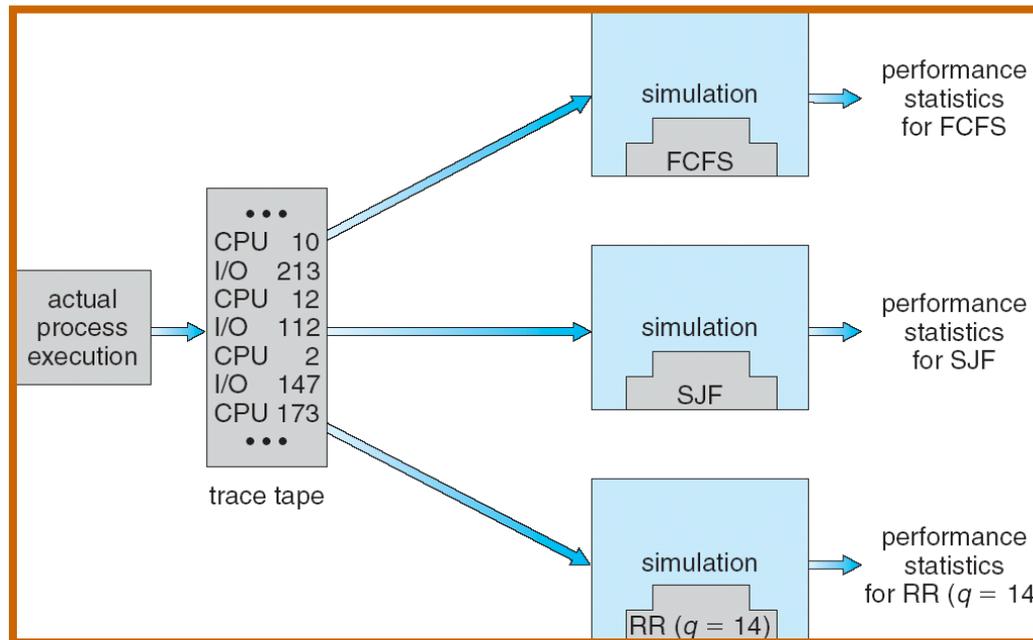
# short jobs/ # long jobs	% of CPU each short jobs gets	% of CPU each long jobs gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

– We
re

- » If load average is 100, hard to make progress
- » One approach: log some user out

How to Evaluate a Scheduling algorithm?

- Deterministic modeling
 - takes a predetermined workload and compute the performance of each algorithm for that workload
- Queueing models
 - Mathematical approach for handling stochastic workloads
- Implementation/Simulation:
 - Build system which allows actual algorithms to be run against actual data – most flexible/general



- Consider mix of interactive and high throughput apps:
 - How to best schedule them?
 - How to recognize one from the other?
 - » Do you trust app to say that it is “interactive”?
 - Should you schedule the set of apps identically on servers, workstations, pads, and cellphones?
- For instance, is Burst Time (observed) useful to decide which application gets CPU time?
 - Short Bursts \Rightarrow Interactivity \Rightarrow High Priority?
- Assumptions encoded into many schedulers:
 - Apps that sleep a lot and have short bursts must be interactive apps – they should get high priority
 - Apps that compute a lot should get low(er?) priority, since they won't notice intermittent bursts from interactive apps
- Hard to characterize apps:
 - What about apps that sleep for a long time, but then compute for a long time?
 - Or, what about apps that must run under all circumstances (say periodically)

What if we Knew the Future?

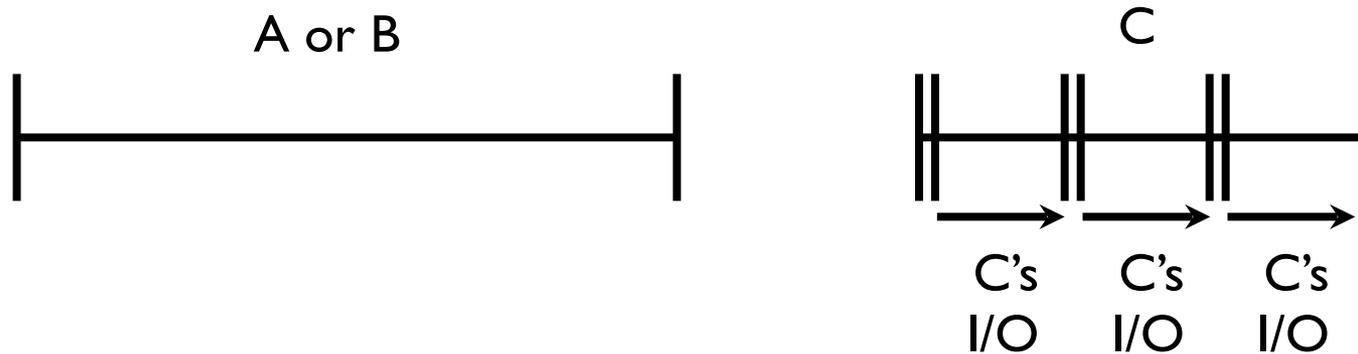
- Could we always mirror best FCFS?
- Shortest Job First (SJF):
 - Run whatever job has least amount of computation to do
 - Sometimes called “Shortest Time to Completion First” (STCF)
- Shortest Remaining Time First (SRTF):
 - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
 - Sometimes called “Shortest Remaining Time to Completion First” (SRTCF)
- These can be applied to whole program or current CPU burst
 - Idea is to get short jobs out of the system
 - Big effect on short jobs, only small effect on long ones
 - Result is better average response time



Discussion

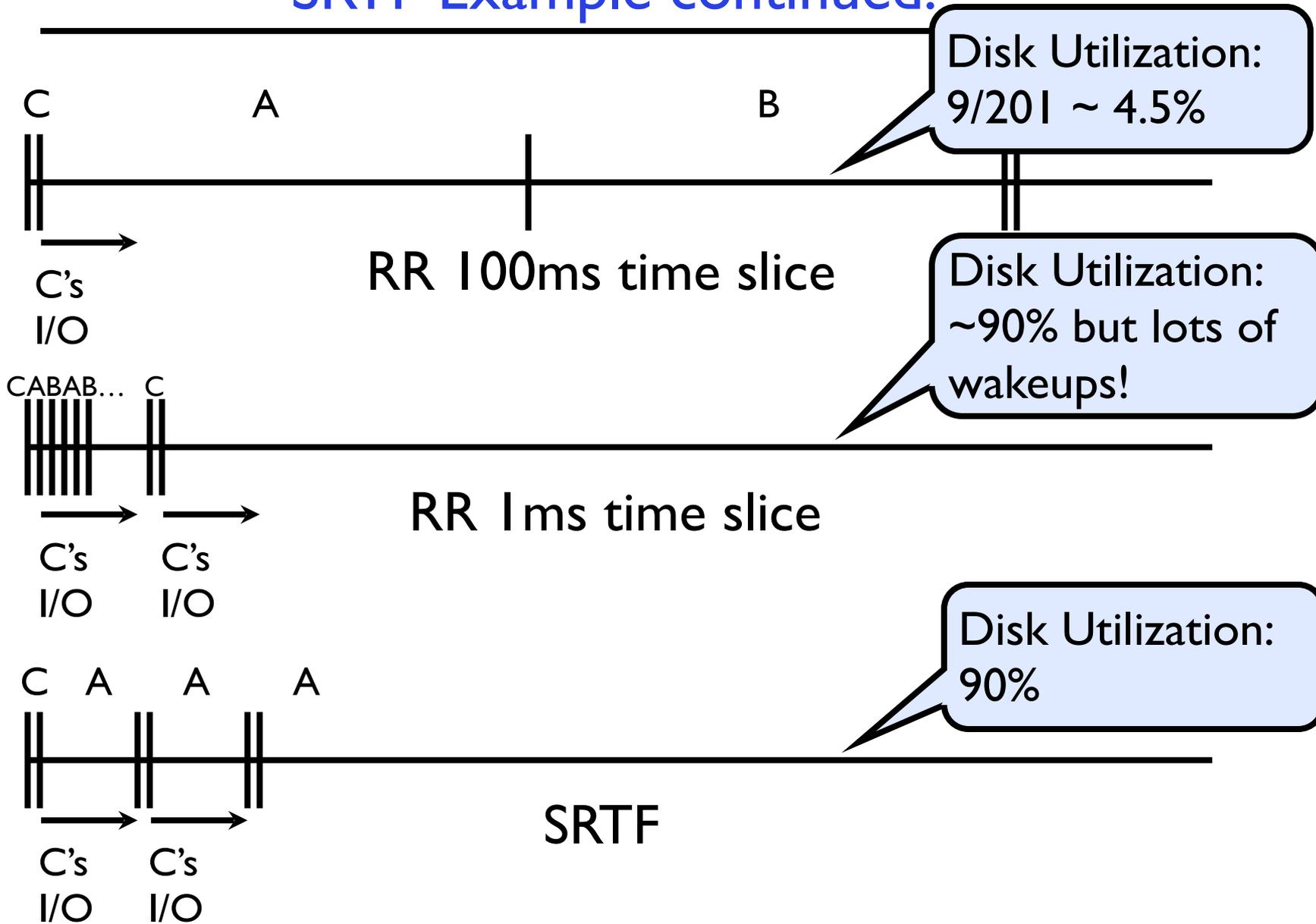
- SJF/SRTF are the best you can do at minimizing average response time
 - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
 - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS
 - What if all jobs the same length?
 - » SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
 - What if jobs have varying length?
 - » SRTF: short jobs not stuck behind long ones

Example to illustrate benefits of SRTF



- Three jobs:
 - A, B: both CPU bound, run for 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 could use 100% of the CPU
- With FCFS:
 - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
 - Easier to see with a timeline

SRTF Example continued:



SRTF Further discussion

- Starvation
 - SRTF can lead to starvation if many small jobs!
 - Large jobs never get to run
- Somehow need to predict future
 - How can we do this?
 - Some systems ask the user
 - » When you submit a job, have to say how long it will take
 - » To stop cheating, system kills job if takes too long
 - But: hard to predict job's runtime even for non-malicious users
- Bottom line, can't really know how long job will take
 - However, can use SRTF as a yardstick for measuring other policies
 - Optimal, so can't do any better
- SRTF Pros & Cons
 - Optimal (average response time) (+)
 - Hard to predict future (-)
 - Unfair (-)

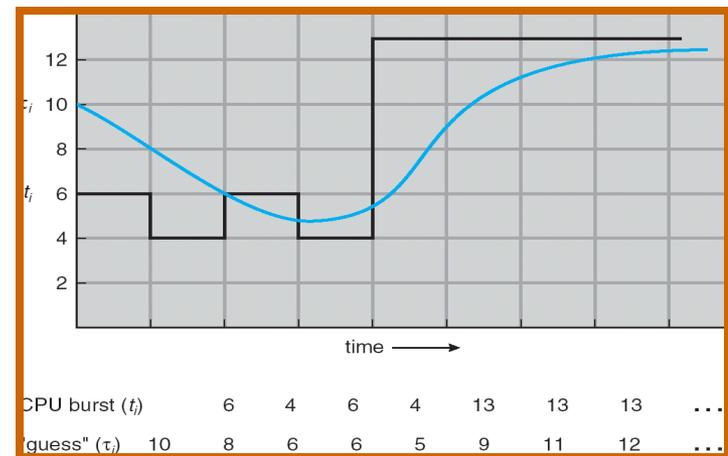


Predicting the Length of the Next CPU Burst

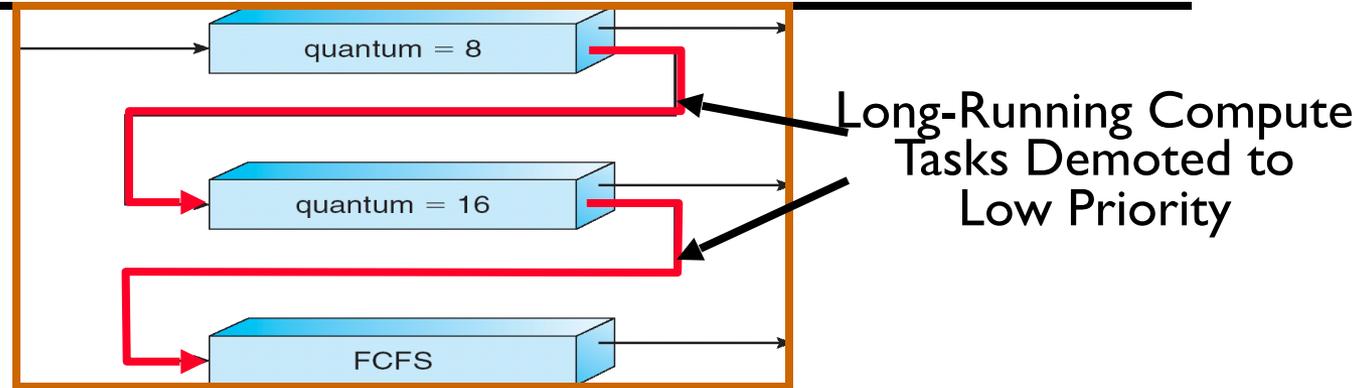
- **Adaptive:** Changing policy based on past behavior
 - CPU scheduling, in virtual memory, in file systems, etc
 - Works because programs have predictable behavior
 - » If program was I/O bound in past, likely in future
 - » If computer behavior were random, wouldn't help
- Example: SRTF with estimated 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)

– For instance,

exponential averaging
 $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$
with $(0 < \alpha \leq 1)$

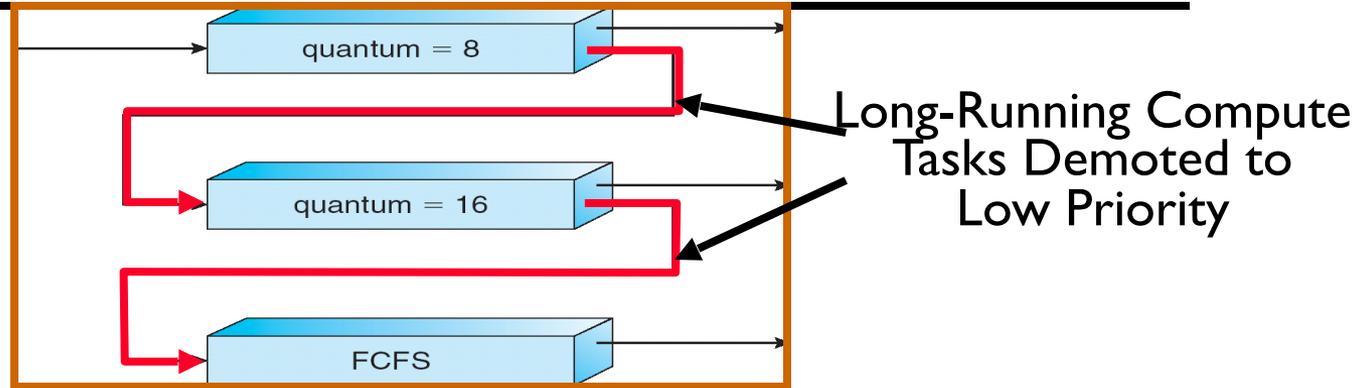


Multi-Level Feedback Scheduling



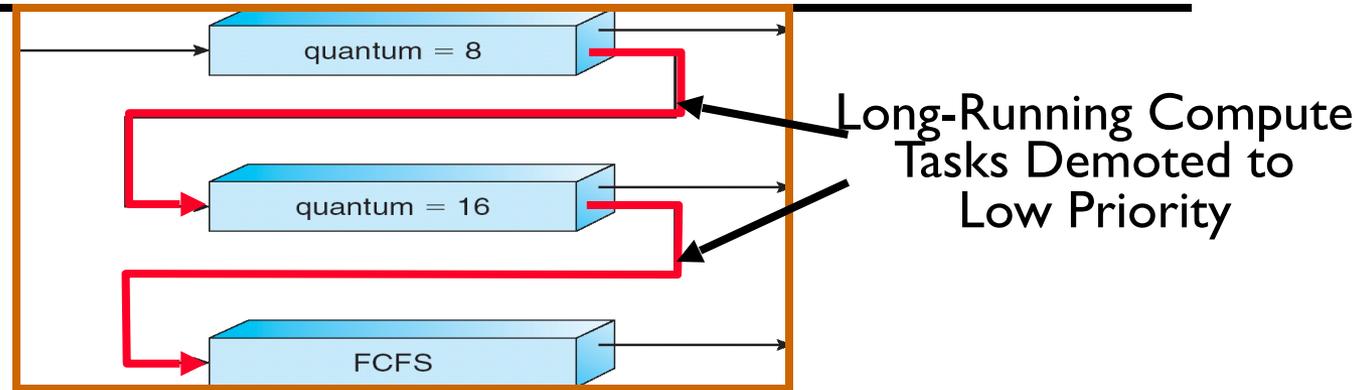
- Another method for exploiting past behavior (first use in CTSS)
 - Multiple queues, each with different priority
 - » Higher priority queues often considered “foreground” tasks
 - Each queue has its own scheduling algorithm
 - » e.g. foreground – RR, background – FCFS
 - » Sometimes multiple RR priorities 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
 - If timeout doesn’t expire, push up one level (or to top)

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.
 - Time slice:
 - » each queue gets a certain amount of CPU time
 - » e.g., 70% to highest, 20% next, 10% lowest

Scheduling Details



- **Countermeasure:** user action that can foil intent of the OS designers
 - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
 - Of course, if everyone did this, wouldn't work!
- Example of Othello program:
 - Playing against competitor, so key was to do computing at higher priority than the competitors.
 - » Put in printf's, ran much faster!

Case Study: Linux $O(1)$ Scheduler



- Priority-based scheduler: 140 priorities
 - 40 for “user tasks” (set by “nice”), 100 for “Realtime/Kernel”
 - Lower priority value \Rightarrow higher priority (for nice values)
 - Highest priority value \Rightarrow Lower priority (for realtime values)
 - All algorithms $O(1)$
 - » Timeslices/priorities/interactivity credits all computed when job finishes time slice
 - » 140-bit bit mask indicates presence or absence of job at given priority level
- Two separate priority queues: “active” and “expired”
 - All tasks in the active queue use up their timeslices and get placed on the expired queue, after which queues swapped
- Timeslice depends on priority – linearly mapped onto timeslice range
 - Like a multi-level queue (one queue per priority) with different timeslice at each level
 - Execution split into “Timeslice Granularity” chunks – round robin through priority

O(1) Scheduler Continued

- Heuristics
 - User-task priority adjusted ± 5 based on heuristics
 - » $p \rightarrow \text{sleep_avg} = \text{sleep_time} - \text{run_time}$
 - » Higher `sleep_avg` \Rightarrow more I/O bound the task, more reward (and vice versa)
 - Interactive Credit
 - » Earned when a task sleeps for a “long” time
 - » Spend when a task runs for a “long” time
 - » IC is used to provide hysteresis to avoid changing interactivity for temporary changes in behavior
 - However, “interactive tasks” get special dispensation
 - » To try to maintain interactivity
 - » Placed back into active queue, unless some other task has been starved for too long...
- Real-Time Tasks
 - Always preempt non-RT tasks
 - No dynamic adjustment of priorities
 - Scheduling schemes:
 - » `SCHED_FIFO`: preempts other tasks, no timeslice limit
 - » `SCHED_RR`: preempts normal tasks, RR scheduling amongst tasks of same priority

Linux Completely Fair Scheduler (CFS)

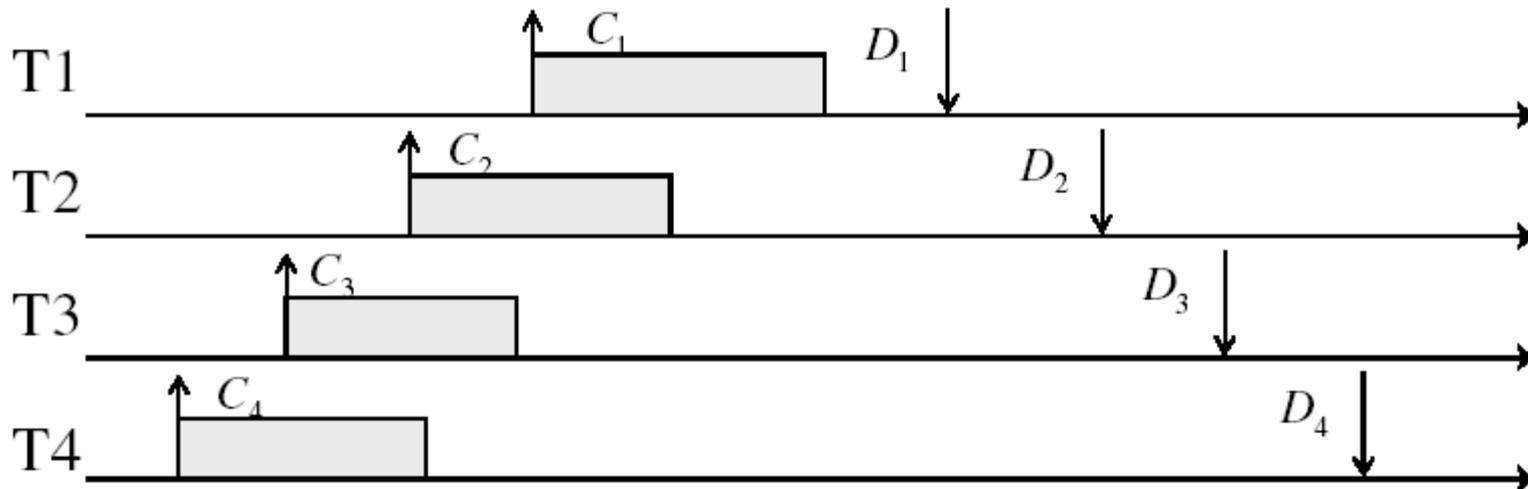
- First appeared in 2.6.23, modified in 2.6.24
- “CFS doesn't track sleeping time and doesn't use heuristics to identify interactive tasks—it just makes sure every process gets a fair share of CPU within a set amount of time given the number of runnable processes on the CPU.”
- Inspired by Networking “Fair Queueing”
 - Each process given their fair share of resources
 - Models an “ideal multitasking processor” in which N processes execute simultaneously as if they truly got $1/N$ of the processor
 - » Tries to give each process an equal fraction of the processor
 - Priorities reflected by weights such that increasing a task's priority by I always gives the same fractional increase in CPU time – regardless of current priority

Real-Time Scheduling (RTS)

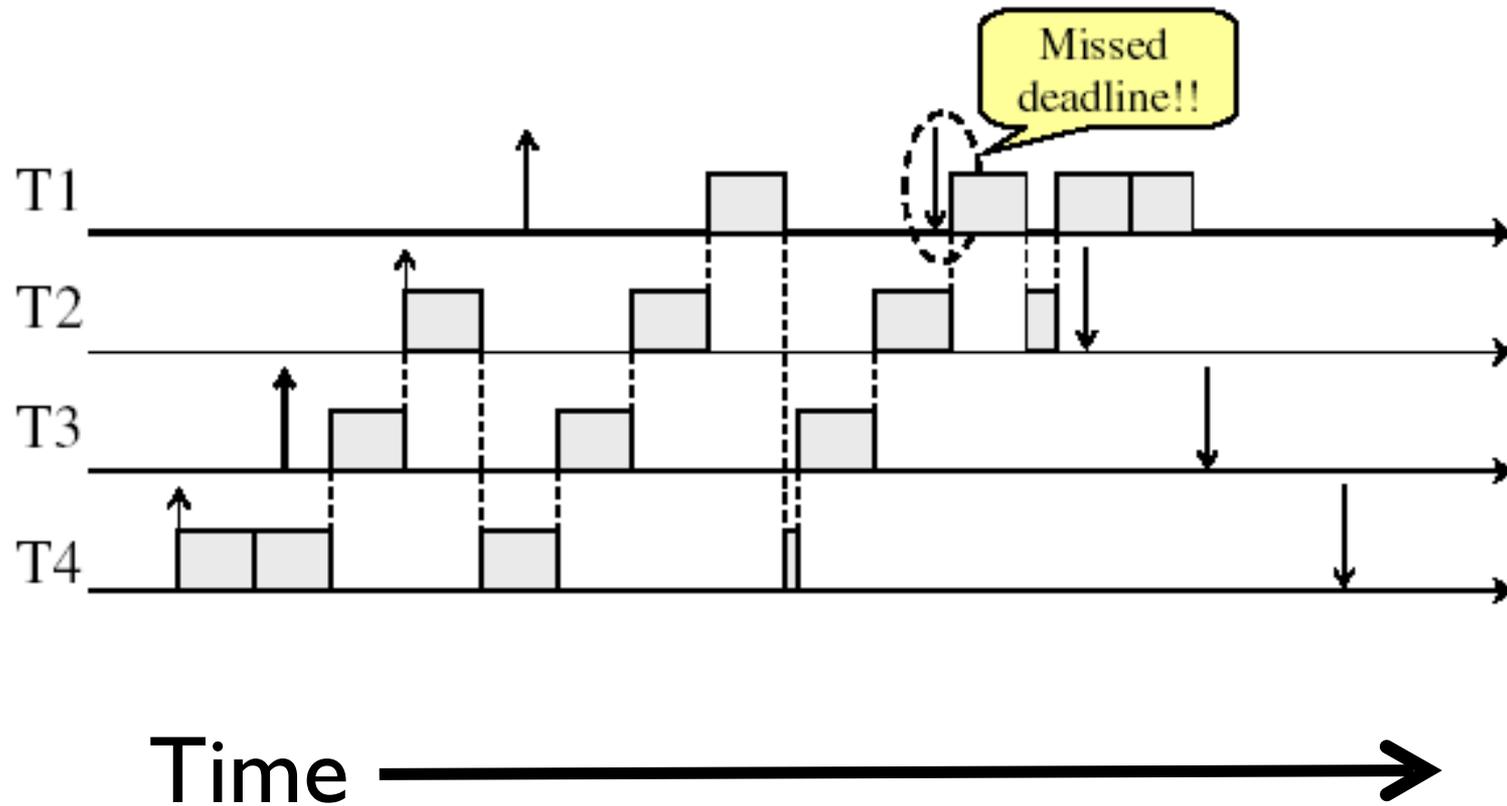
- Efficiency is important but **predictability** is essential:
 - We need to predict with confidence worst case response times for systems
 - In RTS, performance guarantees are:
 - » Task- and/or class centric and often ensured a priori
 - In conventional systems, performance is:
 - » System/throughput oriented with post-processing (... wait and see ...)
 - Real-time is about enforcing predictability, and does not equal fast computing!!!
- Hard Real-Time
 - *Attempt to meet all deadlines*
 - EDF (Earliest Deadline First), LLF (Least Laxity First), RMS (Rate-Monotonic Scheduling), DM (Deadline Monotonic Scheduling)
- Soft Real-Time
 - *Attempt to meet deadlines with high probability*
 - Minimize miss ratio / maximize completion ratio (firm real-time)
 - Important for multimedia applications
 - CBS (Constant Bandwidth Server)

Example: Workload Characteristics

- Tasks are preemptable, independent with arbitrary arrival (=release) times
- Tasks have deadlines (D) and known computation times (C)
- Example Setup:

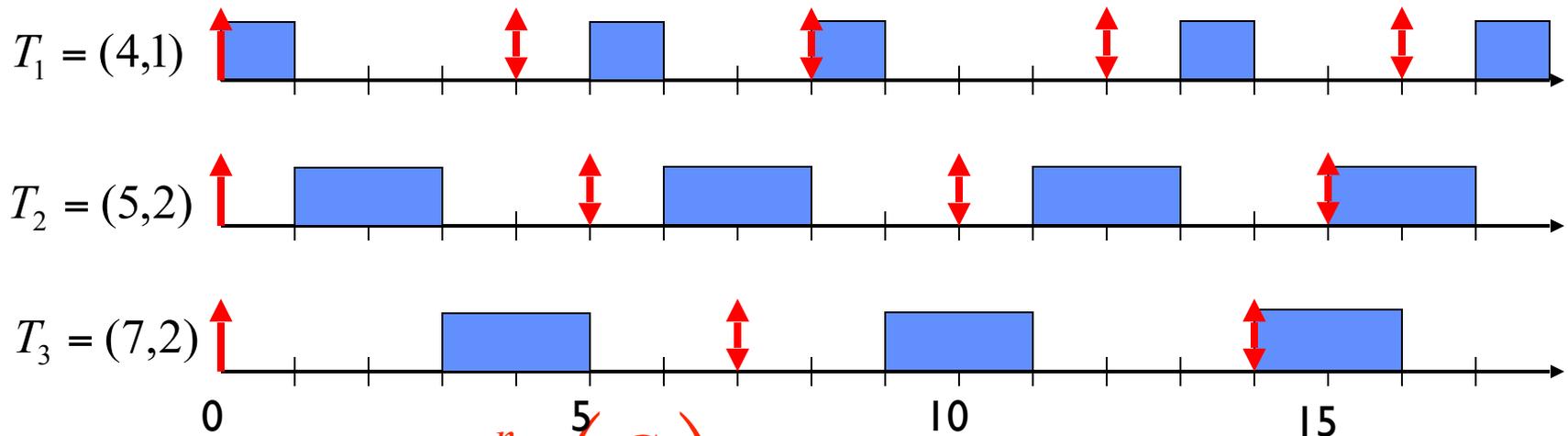


Example: Round-Robin Scheduling Doesn't Work



Earliest Deadline First (EDF)

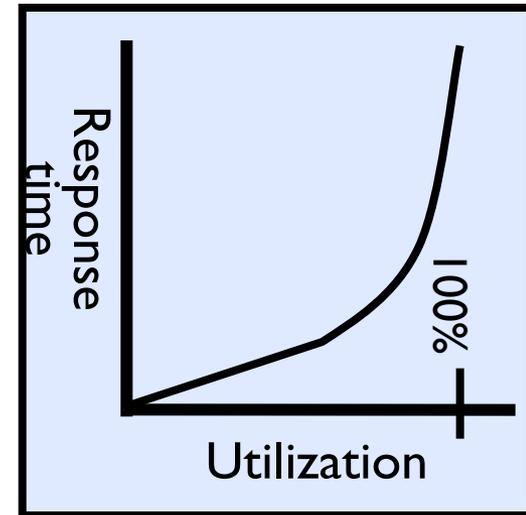
- Tasks **periodic** with period P and computation C in each period: (P_i, C_i) for each task i
- Preemptive priority-based dynamic scheduling:
 - Each task is assigned a (current) priority based on how close the absolute deadline is (i.e. $D_i^{t+1} = D_i^t + P_i$ for each task!)
 - The scheduler always schedules the active task with the closest absolute deadline



- Schedulable when $\sum_{i=1}^n \left(\frac{C_i}{P_i} \right) \leq 1$

A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
 - When there aren't enough resources to go around
- When should you simply buy a faster computer?
 - (Or network link, or expanded highway, or ...)
 - One approach: Buy it when it will pay for itself in improved response time
 - » Perhaps you're paying for worse response time in reduced productivity, customer angst, etc...
 - » Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization \Rightarrow 100%
- An interesting implication of this curve:
 - Most scheduling algorithms work fine in the “linear” portion of the load curve, fail otherwise
 - Argues for buying a faster X when hit “knee” of curve



Summary (1 of 2)

- **Round-Robin Scheduling:**
 - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
 - Pros: Better for short jobs
- **Shortest Job First (SJF)/Shortest Remaining Time First (SRTF):**
 - Run whatever job has the least amount of computation to do/least remaining amount of computation to do
 - Pros: Optimal (average response time)
 - Cons: Hard to predict future, Unfair
- **Multi-Level Feedback Scheduling:**
 - Multiple queues of different priorities and scheduling algorithms
 - Automatic promotion/demotion of process priority in order to approximate SJF/SRTF

Summary (2 of 2)

- **Lottery Scheduling:**
 - Give each thread a priority-dependent number of tokens (short tasks \Rightarrow more tokens)
- **Linux CFS Scheduler: Fair fraction of CPU**
 - Approximates a “ideal” multitasking processor
- **Realtime Schedulers such as EDF**
 - Guaranteed behavior by meeting deadlines
 - Realtime tasks defined by tuple of compute time and period
 - Schedulability test: is it possible to meet deadlines with proposed set of processes?