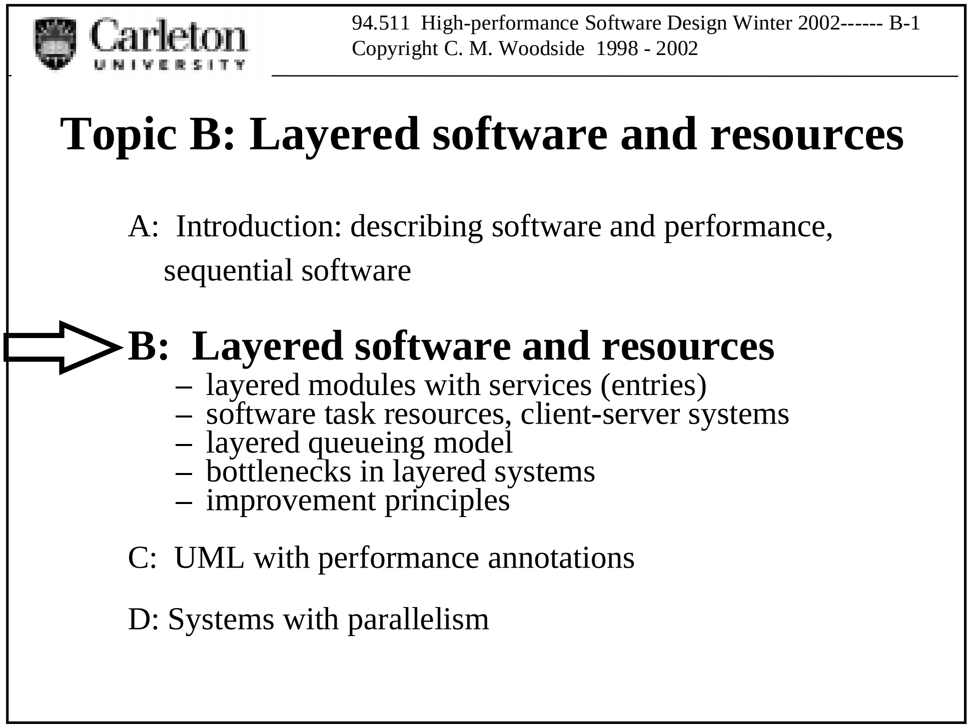


Topic B: Layered software and resources

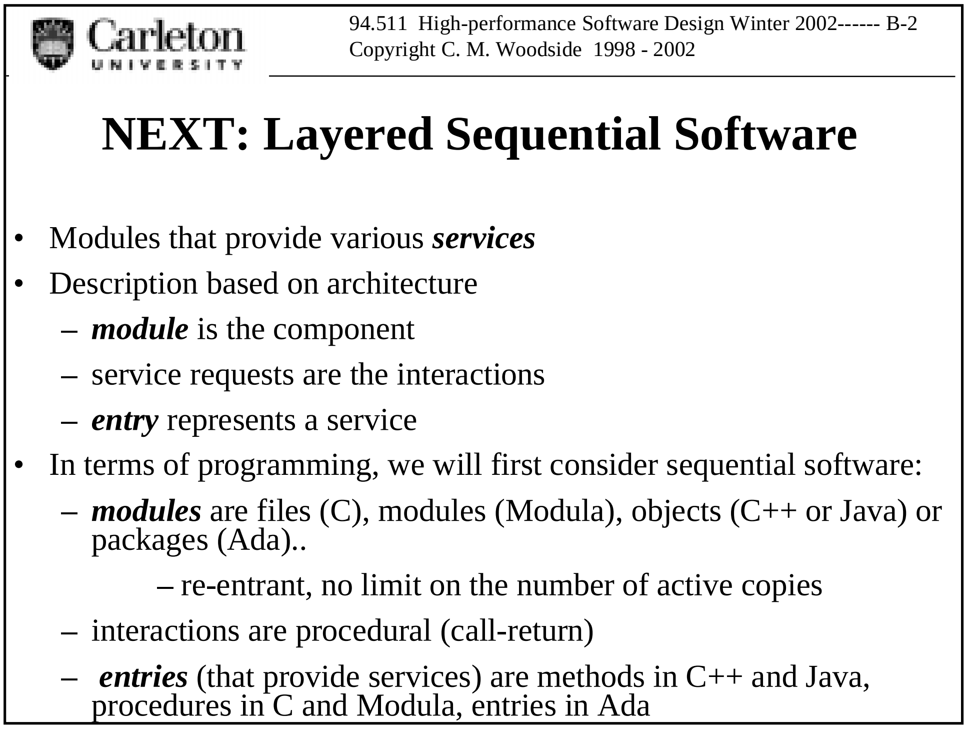
A: Introduction: describing software and performance, sequential software

B: Layered software and resources

- – layered modules with services (entries)
- – software task resources, client-server systems
- – layered queueing model
- – bottlenecks in layered systems
- – improvement principles

C: UML with performance annotations

D: Systems with parallelism

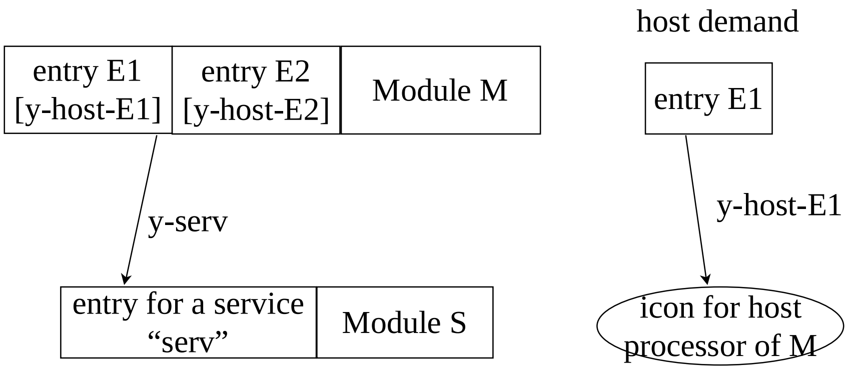


NEXT: Layered Sequential Software

- Modules that provide various ***services***
- Description based on architecture
  - – ***module*** is the component
  - – service requests are the interactions
  - – ***entry*** represents a service
- In terms of programming, we will first consider sequential software:
  - – ***modules*** are files (C), modules (Modula), objects (C++ or Java) or packages (Ada)..
    - – re-entrant, no limit on the number of active copies
  - – interactions are procedural (call-return)
  - – ***entries*** (that provide services) are methods in C++ and Java, procedures in C and Modula, entries in Ada

Module notation

- module, entry, service demands to other modules
- exactly one *host device or processor* for a real software module
  - alternative notation for host demand.......



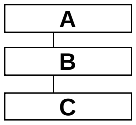
To get the demand parameters of a module model

- direct tests using OS instrumentation to measure resource demands during the test
- profiling C or C++ programs with gprof or quantify
  - CPU demands per call of each procedure
  - procedure demands are totalled over all calls, and also broken down according to the calling procedure
- strace utility in some UNIX systems records all system calls

- start from a scenario for each entry
  - if given a global scenario, divide into a sub-scenario for each entry

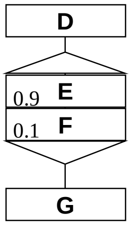
Example: A Module with two services, from their activity graphs,

Graph “serv1”



MeanTimes	CPU M-in	file ops	create ops	inout ops
1	0.1	1.8		
1	0.2		1	
1	0.1			13
Wtd sum	0.4	1.8	1	13

Graph “serv2”



MeanTimes	CPU M-in	file ops	create ops	inout ops
1	0.15			
1	0.01			
0.9	0.2			3
0.1	0.6			6
1	0.1	2.5		
Wtd sum	0.5	2.5		3.3

Module with two services (cont’d): Reduce to device demands...

Suppose the logical operations have the following device demands:

Logical service	cpu M-in	disk ops
File op	0.02	1.3
create	0.75	
inout	0.29	

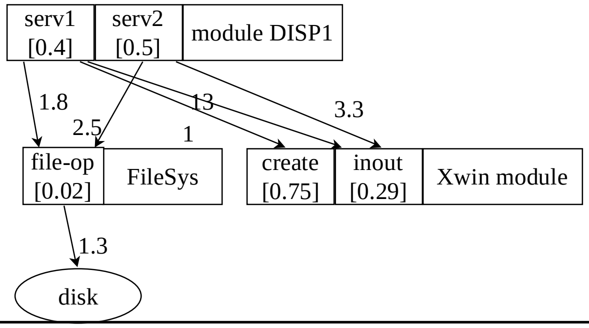
Then we could convert the service demand columns to device demands:

serv1	4.956	2.34
serv2	1.507	3.25

Logical operation demands can be kept external or folded in, to give parameters of one service entry

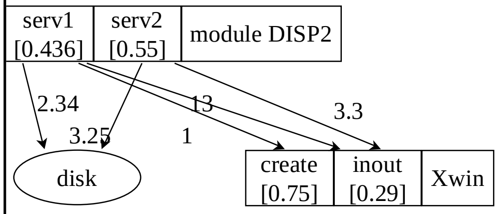
Module with two services... result

- Consider the demands for logical services file-op create and inout, in the original table, as demands to other modules:
  - *FileSys* for the filesystem with entry ***file-op***,
  - *Xwin* for the X-window system with entries ***create*** and ***inout***.
- host demands (CPU) are shown on entries in [square] brackets

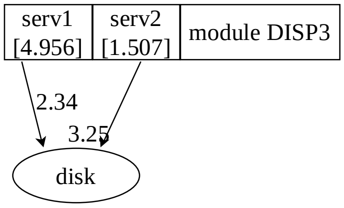


Granularity: Module model aggregation

- there is a lot of choice possible in what is the boundary of a module for the model (granularity of modeling)
- a submodule can be included in the same way that modules are reduced... to get the aggregate demands



II... ***put file-op inside DISP***



III... ***put Xwin inside too***

0.436 = 0.4 + 1.8*0.02;
2.34 = 1.8 * 1.3

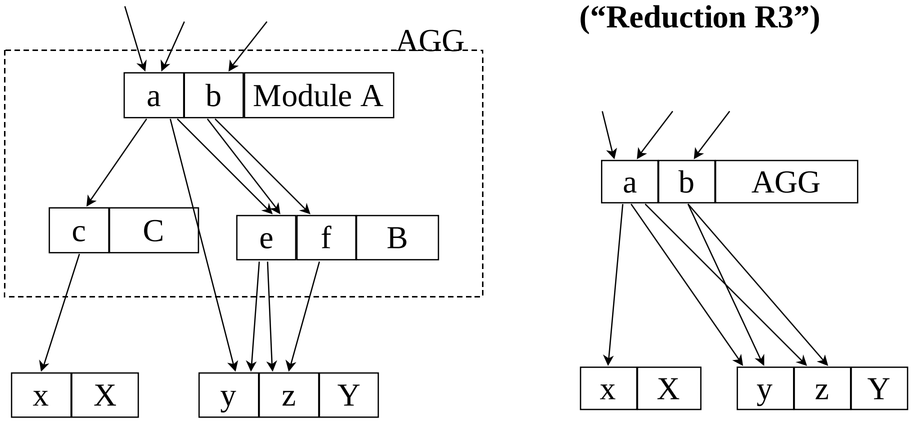
0.55 = 0.5 + 2.5 * 0.02
3.25 = 2.5 * 1.3

4.956 = 0.436 + 0.75 + 13 * 0.29
1.507 = 0.55 + 3.3 * 0.29

General module aggregation

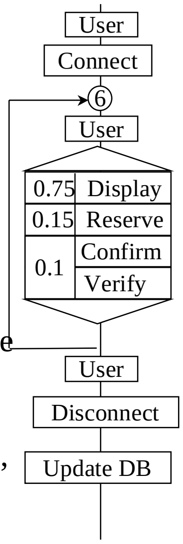
- we may model a subsystem as a module, aggregating its internals
- first case... one internal module A provides the interface

(more complex... requests go to several modules, all form interface)



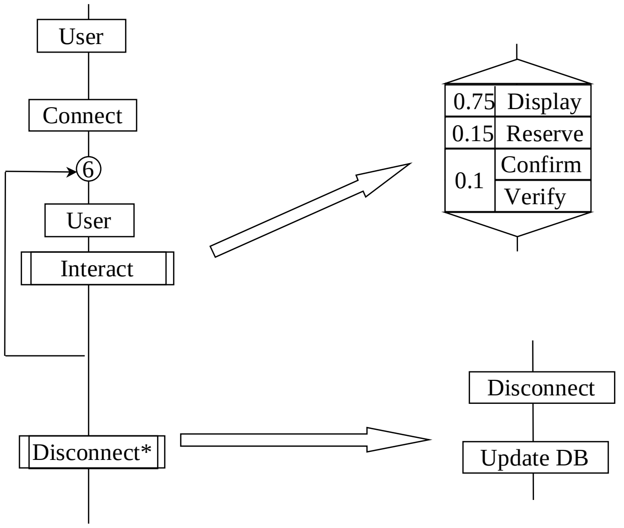
Dividing an end-to-end activity graph across modules and entries: Reservation Server

- subdivide it into separate subgraphs
- analyze each one
- transition from one entry to another becomes a service demand
- Here, suppose a session control module invokes TicketRes each time the user input arrives.
- Allocate “Update DB” to Disconnect



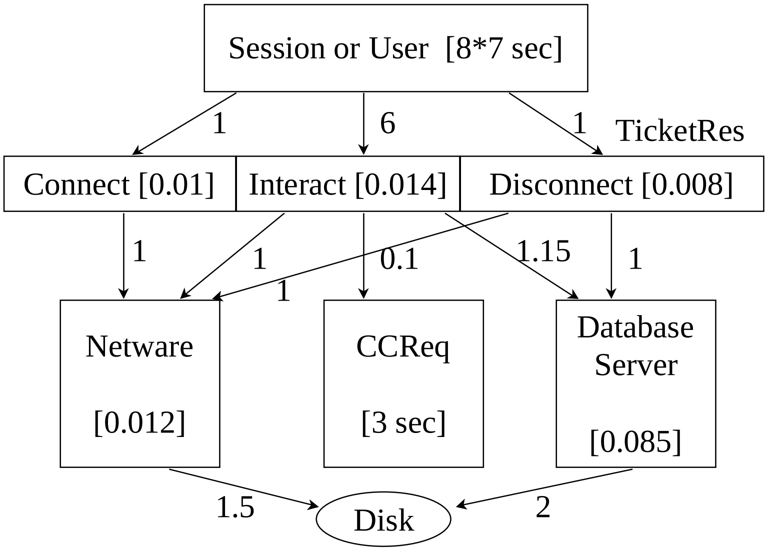
Host	Module	Entry
Srvr	TicketRes	Connect
PC	Session	User
Srvr	TicketRes	Interact
Srvr	TicketRes	Interact
Srvr	TicketRes	Interact
Srvr	TicketRes	Interact
Srvr	TicketRes	Interact
PC	Session	User
Srvr	TicketRes	Disconnect
Srvr	TicketRes	Disconnect

Restructure around choice of entries

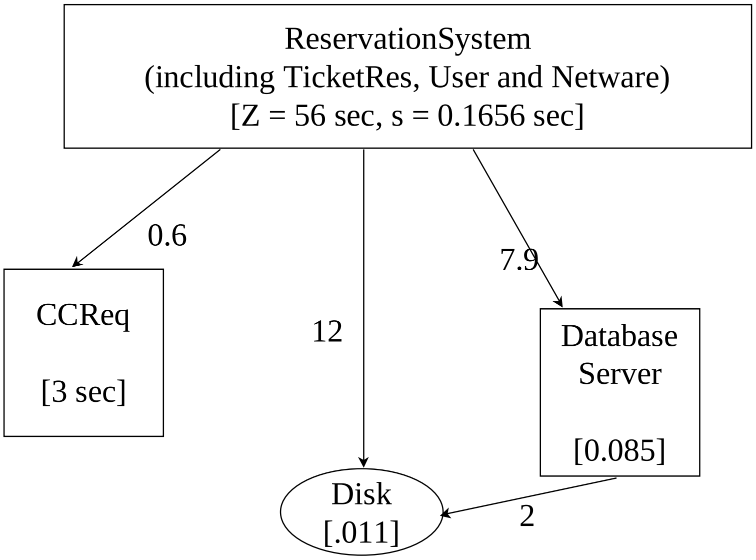


- reduce the parameters of each entry subgraph separately
- “User” is just a 7-sec mean delay, broken out because it takes place elsewhere

Multiple Modules... Reservation System



Aggregate Reservation System



ReservationSystem
(including TicketRes, User and Netware)
[Z = 56 sec, s = 0.1656 sec]

0.6

7.9

12

CCReq

[3 sec]

Database Server

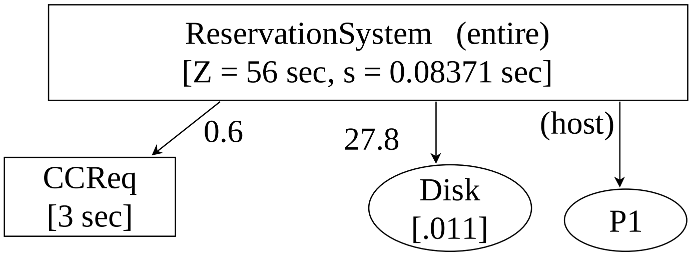
[0.085]

Disk
[.011]

2

Aggregate ...(cont)... options

All on one processor:



ReservationSystem (entire)
[Z = 56 sec, s = 0.08371 sec]

0.6

27.8

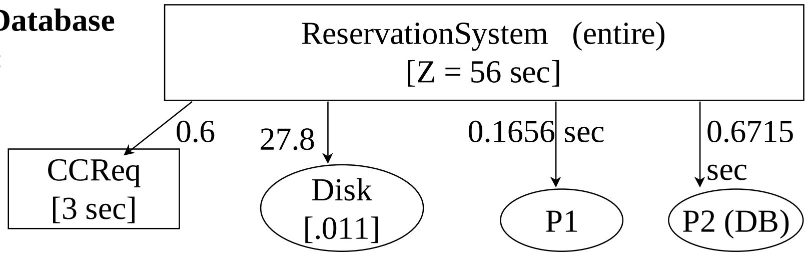
(host)

CCReq
[3 sec]

Disk
[.011]

P1

Separate Database Processor:



ReservationSystem (entire)
[Z = 56 sec]

0.6

27.8

0.1656 sec

0.6715 sec

CCReq
[3 sec]

Disk
[.011]

P1

P2 (DB)

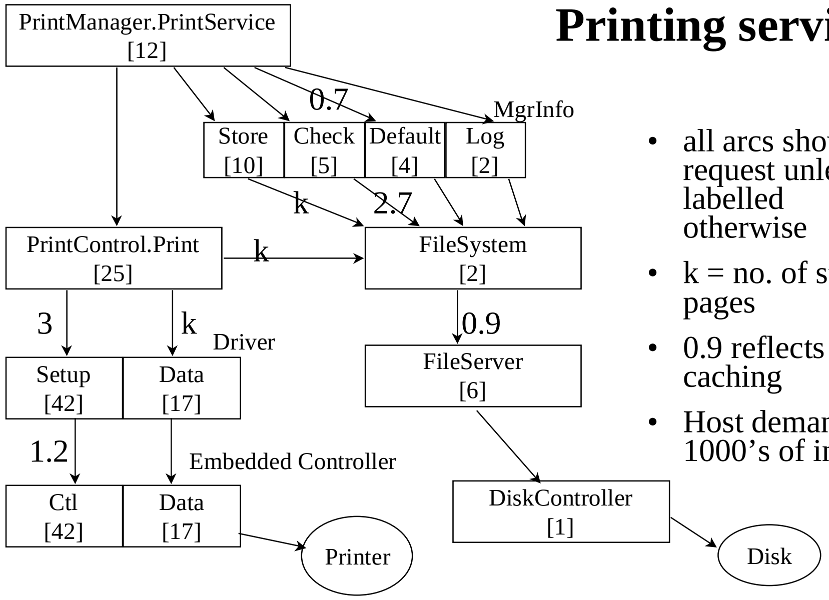
NEXT: Substantial example: Printing Service

- the model is stated from the beginning as a set of modules with entries and demands
  - as might be found by profiling some running components

Features of this example:

- fine-grained modules
- application demand parameters are kept free (values supplied later)
- aggregation of modules
- implicitly, all the modules are re-entrant (there are no resource constraints except devices)... this will be changed later
- initially this will be analyzed as linear software (just sequential re-entrant modules, no parallelism)

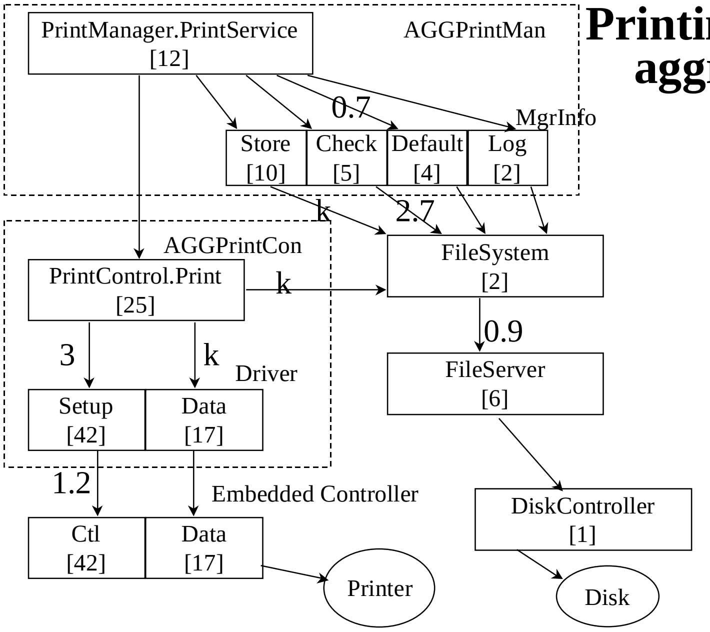
Printing service



- all arcs show one request unless labelled otherwise
- k = no. of storage pages
- 0.9 reflects file caching
- Host demands in 1000’s of inst.

94.511 High-performance Software Design Winter 2002------ B-17
Copyright C. M. Woodside 1998 - 2002

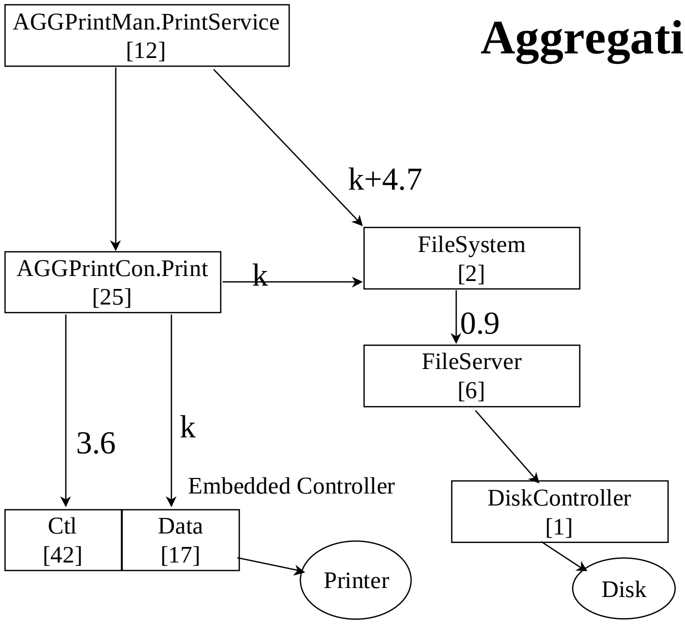
Printing service: aggregation



- AGGPrintMan runs on some workstation
- AGGPrintCon runs on a WS connected to the printer
- Embedded Controller is in the printer itself
- FileSystem module is the local part, running on each WS
- FileServer is the NFS server

94.511 High-performance Software Design Winter 2002------ B-18
Copyright C. M. Woodside 1998 - 2002

Aggregation Results



Printing service: design alternatives and questions

System changes that might occur (scaling, sensitivity)

- different file system (local vs network)
- different file system demands by users,
- different job sizes

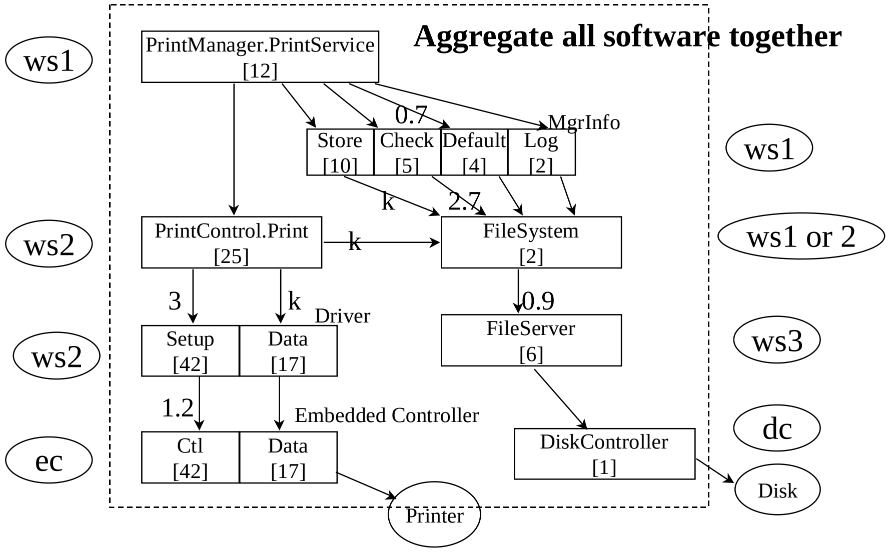
Design alternatives for performance:

- How about multiple printers?
- Can PrintCon prefetch pages?
- Where do jobs queue?... waiting to be picked up by PrintCon
- How about priorities?

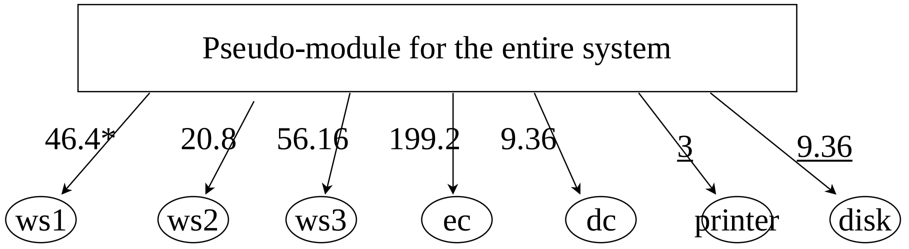
Improving performance using module model analysis

- this is the best understood, most familiar domain for reasoning about optimization
  - no concurrency, so not too complex
  - only “new” idea is to reason in advance using a model
- reduction R4 to device demands of a queueing model, for performance prediction
- direct heuristics on the module model
  - bottleneck identification from the reduction R4: which device has maximum demand?
    - move work away from it
    - substitute replaceable components with others of lower demands
  - identify deep calling patterns with high calling overhead: coalesce modules, or flatten inheritance heirarchies
  - identify frequently used components (as in profiling): “code straightening”; examine details (e.g. activities)

Printing service: Reduction to a flat model (Apply R3 to the entire system to get its device demands)



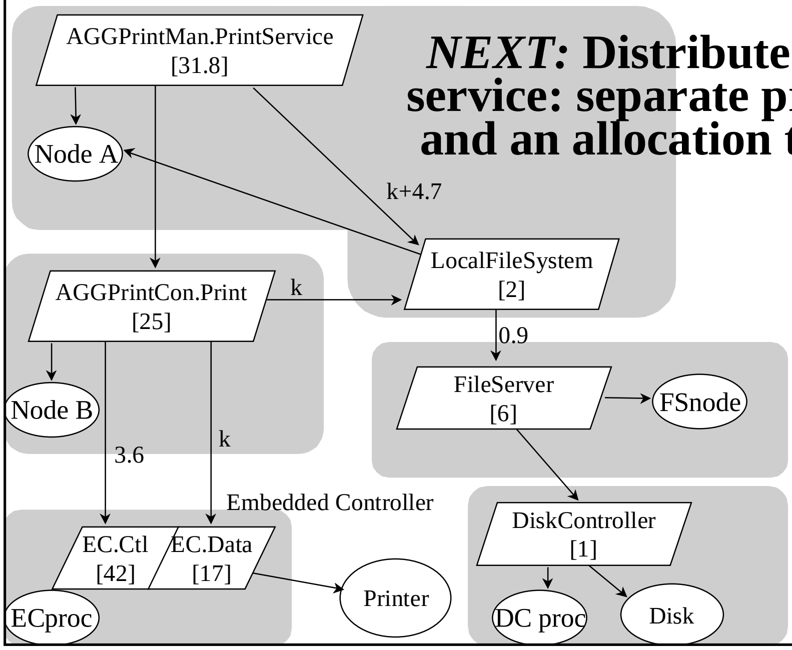
Device workloads...



optime = 1 microsec for all processors................. 0.1 sec 0.014 sec
D = .046 .208 .056 .199 .009 0.3 .131 sec

* request counts for processors are in 1000s of operations

NEXT: Distributed print service: separate processes and an allocation to hosts



Printing service: design alternatives and questions

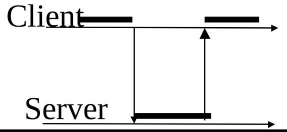
Performance-related questions:

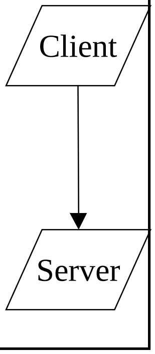
- How about multiple printers?
- Can PrintCon prefetch pages?
- Where do jobs queue?... waiting to be picked up by PrintCon
- How about priorities?

- the “linear” discussion may be too limited to analyze these, so we will now proceed....
  - to a *layered resource architecture* with greater parallelism and concurrency.

NEXT: Layered Resources and Client-Server Systems

- There are multiple programs (***tasks***) running on different hosts
  - operating system processes
- We will consider each program to be a single module... it could be an aggregate of smaller modules... it can include shared libraries.
- “Calls” are now synchronous interactions
  - Client sends request and waits for reply
  - Server executes
  - Server sends reply and then waits for next request
  - Client continues



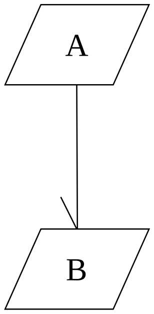


Single-threaded processes

A process or task is an independent autonomous sequential program, with its own program counter and data context.

- it is an OS construct really... UNIX processes are our guide here
- it has a process control block or similar data structure that maintains its context (things like its owner, memory pages, files open, stack pointer, time and resource statistics that you see with the ps command, processes that it has created...)
- using OS facilities it can be created, suspended, activated, and destroyed
  - complex, high cost context switch to suspend/activate (half a millisec approx)
- processes cannot access each others’ data
  - some systems have hardware memory protection
- note that there are some violations in UNIX of complete separation of processes
  - they can share access to a data space outside of each of them,
  - one process can create another; when the parent is destroyed all its children are also
- UNIX daemons are processes that run at set intervals
- the underlying concept of process is more general, for instance it could allow internal parallelism.

An interprocess message
(single-threaded processes, send-and-continue or async)



- Process A sends a message to process B:
  - A executes:

    .....

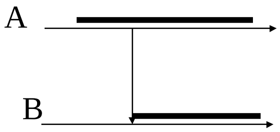
    send(port-address, x-struct)

    ..... continues on
  - B executes:

    .....

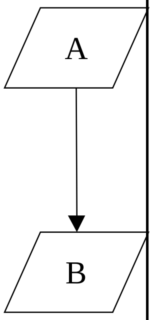
    receive(port-address, y-struct) %it waits for the message

    ..... continues on
- A must know the port-address, the structures must match



- e.g. UNIX socket communication

A synchronous message, or send-and wait



- Process A sends a message to process B, and waits for a reply::
  - A executes:

    .....

    sendwait(port-address, x-struct, reply-port, reply-struct)

    ..... waits for reply, then continues on
  - B executes:

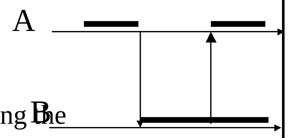
    .....

    receive(port-address, y-struct, reply-port) %it waits for the message

    ..... executes

    reply(reply-port, reply-struct)

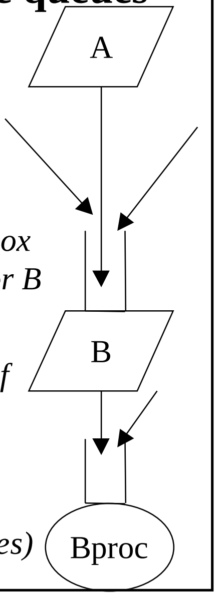
    ...... continues on ***(phase 2)***



- An RPC executes this within the stubs, as well as marshalling the arguments into the x-struct and y-struct
- **B serves A**... and may continue

Interaction via mailboxes and message queues

- Messages may be sent as units, or as parts of a data stream
- if, when B executes *receive*, there is a message, it goes on to complete the receive and continue.
  - – if no message, it becomes blocked on the queue
- When a message arrives for a process which is waiting for a message, the process is put on the ready queue by the OS



queue or mailbox of messages for B

B usually takes the messages in order of arrival (FCFS)

B serves A, and Bproc serves B (layered queues)

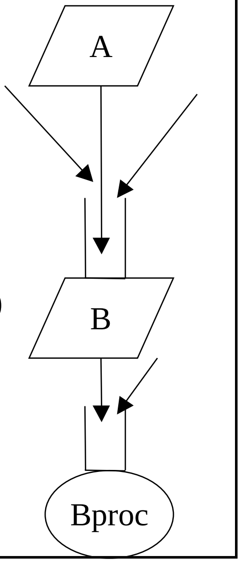
NEXT: Layered queuing for layered service

- A makes a request to B and waits
- A sees a response time

  = (wait for B) + (latency) + (service time of B)
- Service time of B is the time from receiving the message, to time of sending reply

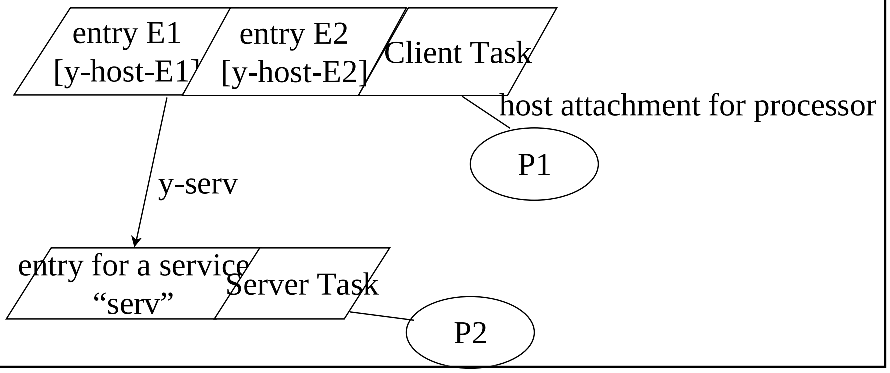
  = (wait for processor)

  + (execution time of processor for demand of B)
- So, the service time of B depends on ***lower layer contention***
- If B makes requests to other software servers their response is also part of the service time of B; this is ***nested service***

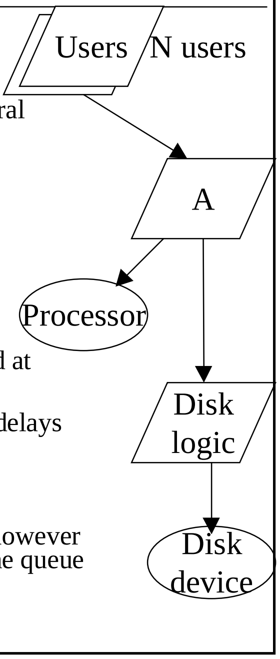


Layered concurrent software

- Concurrent active modules will be called “tasks”, its processor is its host.
- Otherwise it is the same as a component-based model.
- We do not usually aggregate tasks, since they can be allocated separately



Layered queueing model



- ***tasks*** represent OS processes, users and the logic of peripheral devices (e.g. disks)
  - users are pure customers
  - other tasks are both servers and customers
- Workload parameters:
  - each task has a ***host device*** with a ***speed factor***
    - host device is a pure server
  - the entry has a ***host demand “s”*** (for seconds of demand at speed factor 1)
  - the entry has a ***service time x*** which includes its nested delays
  - the entry may also have a ***“think time” Z*** (pure delay)
  - the entry has ***mean demands yi*** to other entries
- a task has a single queue for messages asking for services; however replies to synchronous requests (rendezvous) do not go to the queue (there could be a second mailbox for replies)
- a single-threaded task is a single server

Semantics of basic layered model

- execution time of each entry is broken into “slices”, between requests to other tasks
  - average of Y slices for the entry, Y = 1+ sum yi
  - slice time exponentially distributed (default) , or random with var = CV2 * (square of mean)
- execution path:
  - case of random choice of next request (“stochastic calls”)
    - if for an entry, Y = 1+ sum yi
      - then prob of next request being to entry i = yi/Y
      - prob of ending the entry = 1/Y
    - this gives a random number of requests with mean yi
  - case of deterministic number of requests (“deterministic” calls)
    - exactly yi requests to task i (must be integer)
- synchronous messages block the sender
  - asynchronous messages may also be sent, no blocking or reply

NEXT: Layered model-building

- a layered queueing model is just a module model, *aggregated to the level of the task* (= operating system process)
- a task is just a module with one or more task thread resources attached to it
  - all entry calls are requests, usually inter-task messages to a mailbox or port
  - requests have to queue for the resource
  - a re-entrant module could be modeled by a task with infinite threads; it would have no queueing.
- parameters for host demands and for requests to other tasks are found just as for modules without resources
  - a scenario for an entry: treat other entries as “devices” when reducing the demands
  - a scenario across entries: partition it among the entries, just as for modules
  - module-based demands can be estimated directly by guess or by measurement
    - but, most measurement systems have problems with establishing the context (e.g. the entry)

Layered queueing language (1)

...*a **task*** has a processor, a queue, a discipline and a multiplicity

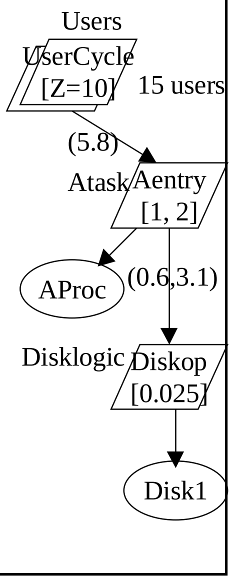
- a ***reference*** task has no requesters; multiple tasks are independent and often run on an "infinite" processor
- servers can be ***multi-threaded***, running on one processor (which can be multiple; one server has one queue to receive requests
- a ***pure server*** makes no requests; devices are pure servers. Every device is modeled as running at least one task which defines its service logic and classes
- a processor can be ***multiple*** (with a single queue). This fits symmetric multiprocessors)

...*a task can have* ***multiple entries:*** workload is associated with entries:

- host demand (mean "s", squared coeff of var'n "c")
- mean requests to other entries (sync "y", async "z", forwarding "f")
- demand is stated by phases; so far we only consider one phase

Layered Queue Language (2): LQNS modeling language

```
G "comment" solver-options -1
P 0     # Processor definitions
    p Pusers f -1 i    # i after -1 means an "infinite" multiplicity processor
    p AProc s -1       # -1 is a terminator for sections, lines or parts of lines
    p Disk1 f -1       # code f for fcfs, s for processor shared
-1
T 0    # Task definitions
    t Users Pusers UserCycle -1  m 15    # multiplicity after -1
    t Atask AProc Aentry -1      # task, its processor, list of entries, -1, multiplicity
    t Disklogic Disk1 Diskop -1
-1
E 0    # Entry definitions
    Z UserCycle 10 -1       # Z for think time
    s Aentry 1 2 -1            # s for host execution demand (phase 1, phase 2)
    s Diskop 0.025 -1
    y UserCycle Aentry 5.8 -1     # mean requests entry to entry
    y Aentry Diskop 0.6 3.1 -1    # mean requests phase 1, phase 2
-1
```



LQNS performance calculations

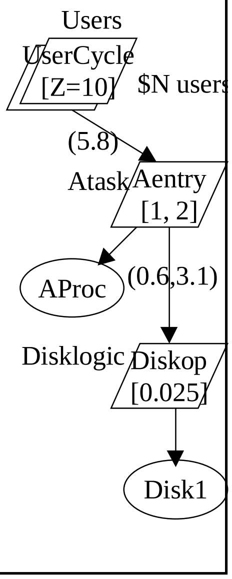
The tool computes

- a set of upper bounds on throughputs based on zero waiting
- mean throughputs of tasks, entries and processors
  - may have to use Little’s result for response times
- mean wait for each entry
- mean service times of entries, (which include nested services and waiting)
- mean utilizations of tasks, entries, and processors
  - (to find bottlenecks)

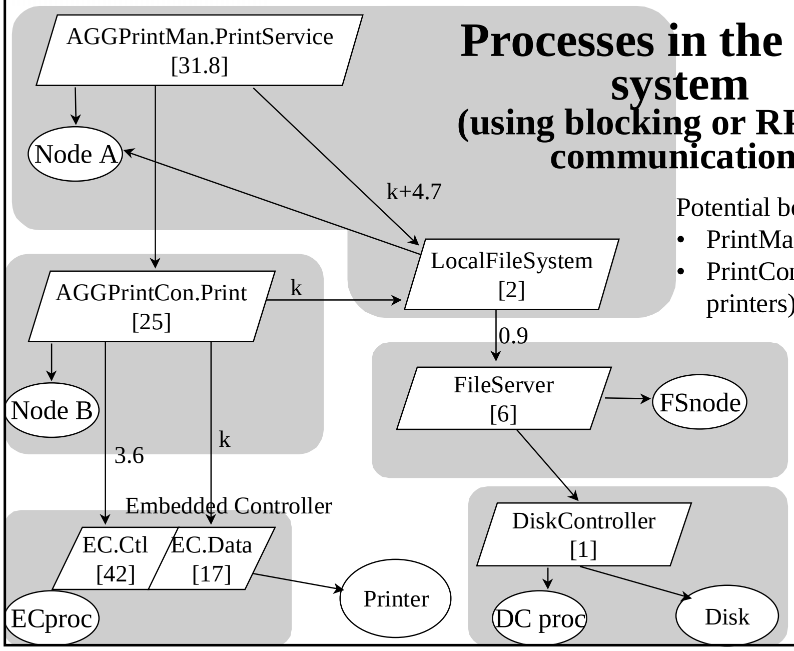
- Use SPEX to do multiple runs over parameter variations

SPEX extended modelling language

```
$N = 1;50;5,100       # initial section sets up values of parameters
$alpha = 1,3,5         # $name are variables, parameters and results
$x1 = 1 - $alpha       # parameters can be defined by expressions
$x2 = 2*$alpha
G “comment” solver-options -1
P 0
.....
T 0
    t Users Pusers UserCycle -1  m $N  %f $Thruput   #instrumentation definition
    t Atask AProc Aentry -1              %u0 $AtaskUtn
    t Disklogic Disk1 Diskop -1          %w  $WaitForDisk
-1
E 0     % Entry definitions
    Z UserCycle 10 -1
    s Aentry 1 2 -1                %s1 $Aph1   # service time of phase 1 of Aentry
.......-1
R 0     # Results section defines what is printed, first is indept variable.
$0 $alpha $N $Thruput $AtaskUtn $WaitForDisk -1
```



Processes in the print system
(using blocking or RPC-like communication)

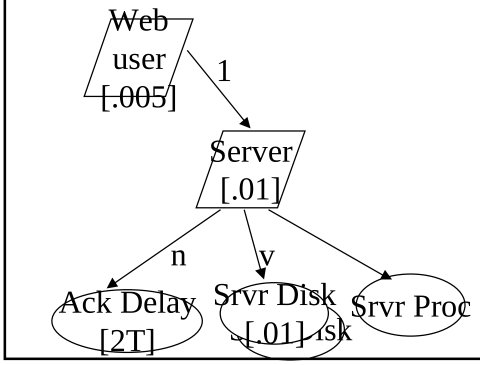


Potential bottlenecks:

- PrintMan
- PrintCon (if multiple printers)

Example: Web Server 3

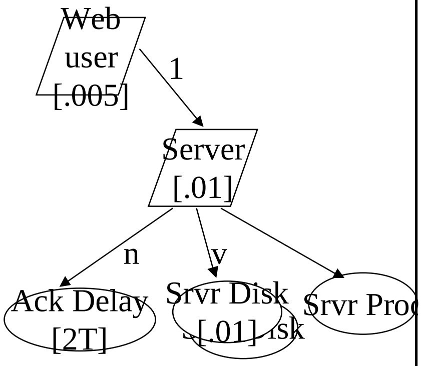
- Server sends long message to the web user
- TCP/IP flow control means the server blocks after sending a “packet”, waiting for acknowledgements, n times
  - represent this as a net delay, then service from the client to receive data and create the ack, then another net delay



Consider a Server with host demand of 10 msec per packet, disk demand of 10 msec spread over many disks, and a long network latency T
... Each time the window fills, the server waits for a round-trip approx.

Web Server Results: need for threads

- A ***single threaded server*** is tied up for the entire network latency, as well as the CPU time plus the disk time
- Thus the service time of one thread is long, and most of that time the ***server CPU and disk are idle***
- We want to carry out ***other services***, during the ack delay
- The mechanism is ***multithreading of the Server***. Each thread babysits one request, including its data movement
  - maintains the user context



Task bottleneck in web sever

- if there is just one copy of the web server (one thread), and it has a service time made up of
  - CPU .005 sec
  - disk .01 sec
  - ack delays of 4sec
  - total 4.01 sec
- this imposes a throughput bound of 1/4.01 requests per sec on the server.

NEXT: Threads.....User threads

- in UNIX, a library lwp (light weight processes) acted like a mini OS within a process, allowing separate threads of control to execute, fork, join, synchronize and communicate.
  - threads can have their own local variables, created when the thread is created
  - threads share a common data space and can communicate through shared variables
  - includes a thread scheduler which could put different priorities on threads. Cheap context switch
- Multi-threaded server:
  - threads may be created when needed, or a pool of threads can be created and made to wait until they are needed
  - listener thread receives a request, creates and dispatches a worker thread to serve it.
    - or dispatches a worker from a pool of static threads
    - can use different threads for different service requests (“entries”), or one thread class
- BUT: any kernel call which blocks the process, blocks all threads (e.g., disk I/O or RPC)

Kernel threads in user processes

- Solaris threads, for instance
  - big difference is the kernel schedules the threads, and while one thread is doing a kernel call such as I/O, others can run,
  - thus the server can have many services underway at once, in cases where the server threads spend a lot of time blocked
  - also, threads can be dispatched by the OS on a multiprocessor to different processors, so can run in parallel
- kernel threads are essential for performance, user threads only provide a form of modularity (and maybe priorities)
  - threads can do RPCs to other servers, as well as disk I/O.
  - in an RPC, the thread retains the context of the service request until the remote service completes
- context switch cost is less than HW process, more than user thread
- other thread systems may be structured differently (Mach, NT)
- Examples: Web servers need threads to avoid serving user requests to completion, one at a time. NFS servers. Most OS kernels have non-blocking threads inside.

Data threads, or virtual threads, or “asynchronous processes”

- a special high-performance approach to writing servers
  - one single threaded process: no context switches of any kind
  - process handles every message separately, requests and acknowledgements, executes a “slice”, sends a message in reply or for some nested service
  - no RPCs... just asynchronous messages
    - however, when a nested service is called, the context of the current service must be retained
    - use data tables to maintain the state of services begun but not completed... hence *data threads*
    - or else store the state in the message, like a token carrying the request around the system (large-grain data flow computing)
  - make no blocking calls to OS... instead, program a separate thread pool or multithreaded process to handle these (this is not a terrific solution, since context switching comes back)
- most expensive for programming, least overhead for context switching
- Examples: DMS call processing software protocol software.

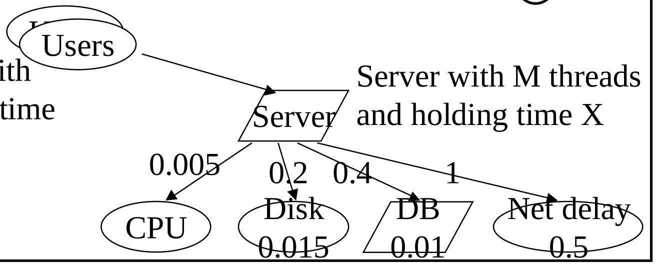
Infinite threads as a model construct

- an infinite “threaded” task has no fixed thread pool or limit on the number of threads
  - could represent a system that creates a thread per request
  - could represent a re-entrant library procedure shared by all users, each executes its own copy
  - some operation with *no limit on the number of concurrent copies*
- can also represent a logical operation which has no execution of its own, but encodes a pattern of requests to services
  - to represent the operations in a scenario, it there is no controller process
  - to represent flow down a pipeline

Results for the web server with net delay

N users	500	500	500	500	2000	2000	2000	2000
M threads	10	30	100	inf	10	30	100	inf
X server	.512	.52	.52	.52	.512	.515	.55	4.99
f thruput	19.5	58.2	90.6	90.6	19.5	56.7	180	200
W user wait	20.6	3.6	0.51	0.5	97.6	29.4	6.1	5
U server	10	30	47	47	10	30	100	1000
U net	9.7	29.1	45.3	45.3	9.7	29.1	90.2	100
U CPU	.097	.29	.45	.45	.097	.29	.90	1.0

N Users with
a thinking time
of 5 sec.



Server with M threads
and holding time X

Web server bottleneck analysis

- D-userCPU = 0.005 and anyways each user has one, so this doesnt constrain the system (it is part of Z if closed system)
- D-SrvrDisk = 0.01 v sec, D-SrvrProc = 0.01 sec
- Z = 4 n sec for the ack delay
- suppose there are 20 disks taking the load equally, and v = 10, so each disk sees D-disk = 0.005 sec
    - then Dmax is 0.01 at the SrvrProc
- One service has a path length of Z + Σ D = .11 + 4n sec

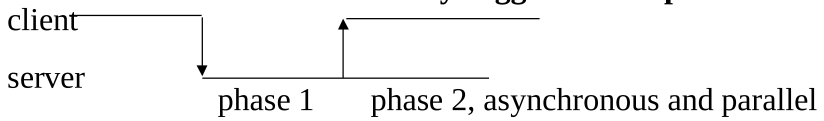
  (this keeps one thread busy all this time)
- SrvrProc utilization over this period averages .01/(.11 + 4n)

  (so if n = 1 this is 1/400)

  (at processor saturation there will be 400 active services!
- This must be provided by multiple *threads* at the server.

NEXT: Concurrency enhancement by "second phases"

Performance Enhancement by Aggressive Replies

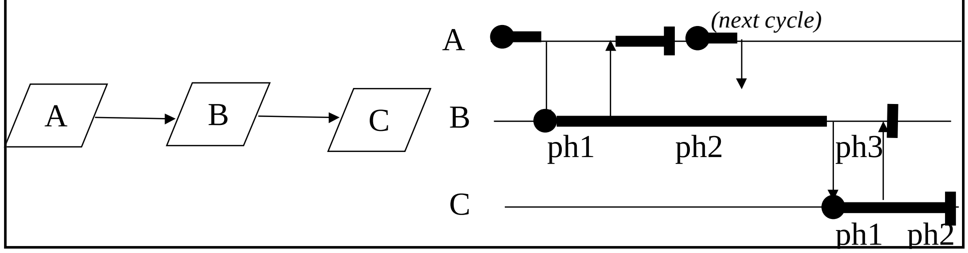


- Idea: Give a reply as early as possible
  - Do postponeable work after the reply, as phase 2
- E. G.: Database server update operation:
  - write to log file before returning, execute final writes later.
- Second-phase model may
  - (a) place this work right after the return (approx), or
  - (b) send a message to a clean-up process that does it later

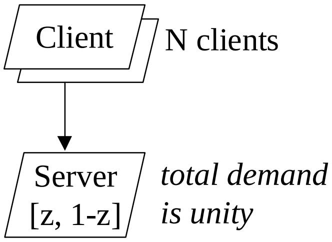
Second and later phases

Also good for:

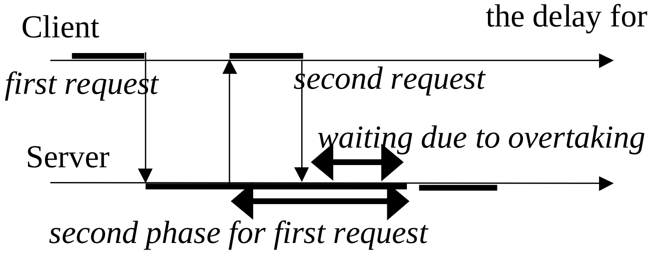
- Task and processor overhead (next receive, scheduling)
- Database delayed operations, NFS delayed writes
- etc.
  - Synchronous pipeline (an example with 3 phases):



Two-phase servers Experiment



- Server does part of its work after the reply
  - Performance 99 paper on "early replies"
- "Walking server" queue for Poisson arrivals has an exact solution
- for networks we approximate the delay for "overtaking"



Impact of Phase 2 on Performance

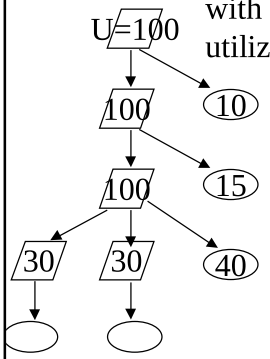
Response times

	z	nusers=1	4	7	10	15	20
All phase 2	0	**0.166**	**1.125**	3.06	4.84	9.90	14.93
	0.2	**0.310**	**0.726**	2.69	4.274	9.473	14.54
	0.4	**0.464**	**0.827**	**1.508**	**3.895**	9.228	14.32
	0.6	**0.629**	**0.996**	**1.792**	**3.981**	9.228	14.31
	0.8	**0.807**	**1.269**	**2.26**	4.492	9.504	14.54
All phase 1	1	0.999	1.642	2.92	5.22	10.03	15.01

Throughputs

	z	nusers=1	4	7	10	15	20
All phase 2	0	**0.193**	**0.652**	0.867	1.015	1.006	1.003
	0.2	**0.188**	**0.698**	0.909	1.078	1.03	1.02
	0.4	**0.183**	**0.686**	**1.075**	**1.124**	1.05	1.03
	0.6	**0.177**	**0.667**	**1.03**	**1.113**	1.05	1.03
	0.8	**0.172**	**0.637**	**0.962**	1.053	1.03	1.02
All phase 1	1	0.166	0.602	0.883	0.978	0.997	0.999

Multiple levels: saturation spreads upwards

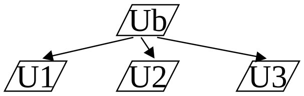


Servers labeled with their utilizations

- Server utilization =
  - the fraction of time it is busy and blocked (one thread)
  - thruput x (thread service time) if one or many threads
  - which is also the *mean number of busy threads*
- The third level server is saturated because it blocks and waits for other servers
- it saturates the layers above it

Software bottleneck: general properties

- Typically the bottom resource is a processor... a bottleneck there is of the familiar hardware type
  - a saturated processor can “push up” on higher resources and make them saturated too, by blocking them
  - this is still a hardware bottleneck
- A pure Software Bottleneck appears only at a higher resource in the layering; lower resources are not responsible
  - typically associated with a “fan-out”... the higher resource delays are contributed by two or more lower resources,
  - BS >1 and $U_b \sim 1$ (Neilson..., IEEE Trans on Soft Eng’g 1995)

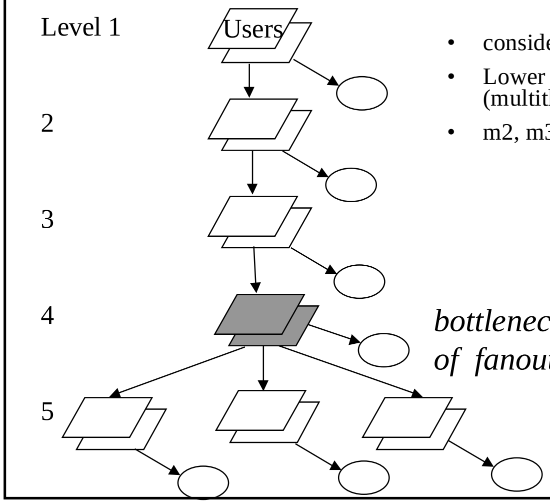


U_b = utilization of task or resource b

- Bottleneck strength measure at task b: $BS_b = U_b / \max_i U_i$

Software bottlenecks

in the “Tower” Deeply Layered Pattern

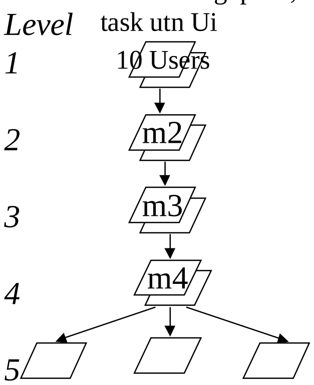


- consider: N users at top layer
- Lower layers may be multi-servers (multithreaded)
- m2, m3, m4 threads at layers 2,3,4.....

bottleneck shows up first here because of fanout (all layers have some fanout)

Software bottleneck relief by multithreading

User throughput f,
task utn Ui



(*single servers at the bottom)*

(*1 sec demand at each server, one request to each lower task)*

A thread is like a clone, dispatched from the same request queue...

...a multi-threaded server behaves like a multi-server; two threads can execute in parallel. *If they are sequentialized by their processor servers, that appears as waiting*

<u>User throughput f depends on thread levels m2, m3, m4:</u>

(m2, m3, m4)...	f ,	(U2, U3, U4, U5)
(1, 1, 1).....	0.166,	(1., 0.83, 0.67, 0.167)
(2, 1, 1).....	0.200,	(0.96, 1, 0.8, 0.2)
(3, 2, 1)....	0.223,	(2.9, 1.64, 0.89, 0.22)
(6, 5, 4)....	0.475,	(5.5, 3.9, 2.75, 0.475)
(10,10,10)..	0.65,	(9.3, 7.8, 6.2, 0.65)

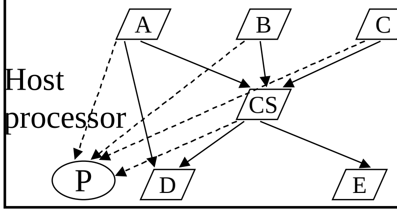
Bottleneck strength at a task =
<u>Utn of task/multiplicity</u>
Max utn of ITS servers and processor

Layered model of a critical section or exclusive lock

- The critical section or lock is modelled as a pseudo-task, which executes the demads of its users (if they all do the same thing, as in a monitor)
- if the users all do different things, it transmits the demands of its user to a separate shadow task that represents the user execution.
- Thus, the critical section pseudo-task is busy while the CS is occupied, and other requests wait in its queue
- a counting semaphore is represented as a CS *multiserver* pseudotask
    - lets a given number of tasks pass, and then queues them
    - implements window flow control on a set of operations

Resource-architecture pattern for a critical section or mutex

- when a set of concurrent tasks share a critical section it limits their concurrency
    - the critical section is a resource
    - treat it as a *pseudo-task* resource: execution is done in the context of the CS
    - workload is moved from the *client* to the *CS pseudo-task*
- when all the tasks execute on the same processor and execute the same code in the crticial section, *it can be modelled as one task:*



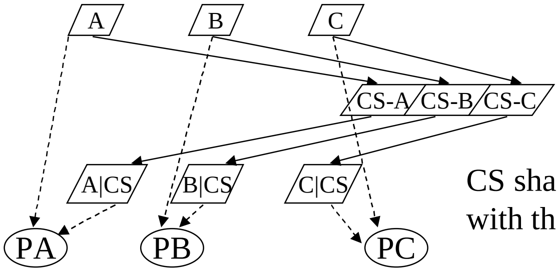
concurrent tasks (users of the CS)

CS pseudo-task, with workload parameters of the CS execution

services invoked during the CS execution

Critical section or mutex...

- we will interpret “critical section” to include mutexes and semaphores and any situation where there is a logical resource
- if the client tasks are on different processors then the CS is modelled in two stages:
    - a pseudo-task CS for mutual exclusion, with a pseudo-host
    - separate entries CS-A... to point to the actual work
    - a *shadow task A|CS...* for each client, to represent the work carried out and the processor attachment of the client



CS pseudo-task with an entry for each client

CS shadow task for each client, with the workload in the mutex

Locks

- a single exclusive lock is like a mutex or CS, however in general locks are different
    - locks are associated with data
    - there may be many separate locks for separate granules of data
    - read or write locks
        - “write” locks are exclusive
        - “read” locks are shared
    - a request may be for a set of several locks at once, covering the data required by the operation
- in distributed systems one solution is a “lock server” which grants the required locks as the first step in a transaction

NEXT: Principles for layered systems

- resource utilization balance
  - try to balance the utilization of physical resources
  - lower layers may require lower utilizations (because delays propagate upwards)
- caching in distributed systems to avoid access conflict and latency
  - can be seen as a replication of the data
- partitioning and replication
  - replication uses and overheads

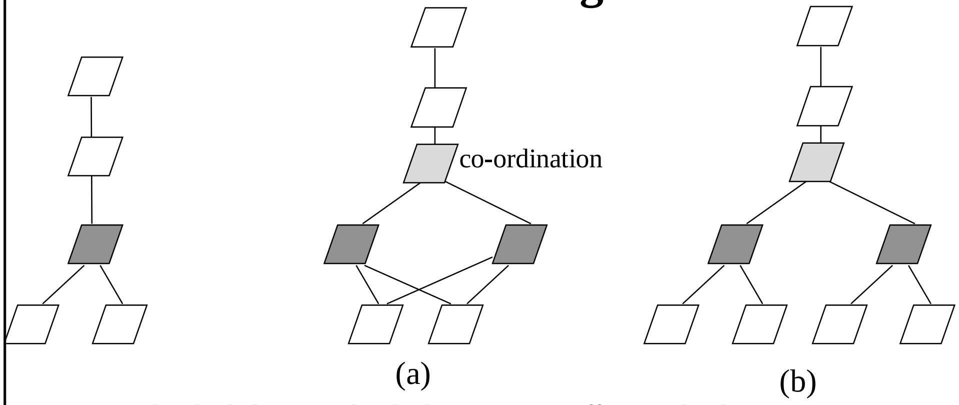
Partitioning and Replication

- To break a bottleneck, replace one server by n servers
- partitioning:
  - divide up the data and/or responsibilities of the server
  - each client goes to different servers, for different functions
    - cost: may need a router to find the particular service needed...
    - cost: may need coordination if multiple partitions must be used and the results combined!!
  - double win if a group of clients goes mostly to just one server
    - (e.g. in geographical partitioning of data)
- replication:
  - duplicate everything in each of the n servers
  - each client goes just to the closest replica
    - biggest win where access costs are much cheaper to the replica
  - cost of coordination of replicas (e.g., update propagation)

Partitioning and replication...

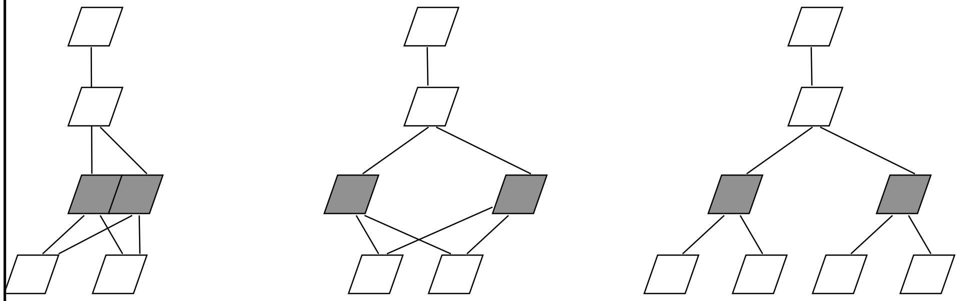
- internet scalability is almost entirely due to partitioning and replication
  - partitioned routing, through heirarchy
  - replicated name service
  - alternative routes (form of partitioning)
  - web mirror sites (form of replication)

Partitioning



- partition the shaded server: divide the services it offers, or the data
  - add coordination layer to select one or both
  - in (a) the partitions share all lower services
  - in (b) the lower services are replicated or partitioned too
    - e.g., separate file server for a local data base

Partitioning by entries



- the shaded task has two entries (two services)
- pull it apart by making a task for each entry
    - relatively painless for the clients because the access logic is already in place

Partitionable tasks

- operations have to be dividable, roughly equally in workload
- OR data must be dividable, roughly equally in access frequency

in such a way that most requests are met by one partition

- natural partitions:
    - most requests to just on(or two) subsets of data)
    - or to one service function
- poorly partitionable
    - data is shared betwen partitions (needs coordinated updates)
    - natural partitions are unbalanced in workload.

Uses of an ORB to glue partitions together

- the ORB essentially knows about services
- provided it can distinguish the partitions by the specification of the service requested, the ORB can determine which partition to use
- then partitioning could be introduced and adapted over time

Partitioning

- departmental data bases
- segmented memories
- RAID disk arrays
  - a case where a request goes to all partitions in parallel
  - partitions are coordinated to provide redundancy and error control
- federated multi-databases
  - basis of three-tier client server systems; middle tier act as coordinating servers as well as business applications

- makes the server smaller and maybe simpler, but makes the client more complex (to decide which partition to use)

A Partitioning Principle

- Partition a task if
  - its operations and data are partitionable
  - partitions can be hosted separately
  - cost of accessing the separate partitions is low enough to give increased capacity

Partitioning and parallel access

- If data is partitionable in terms of updates but most queries access most of it, it can still be partitioned
- … and if the queries can be launched in parallel

- the results must be combined
  - this is a big topic in distributed databases (e.g. parallel join)

Replication

- identical copies of the service and/or the data are kept on multiple sites
  - a client uses any copy and sees the same service
- the replicas together manage consistency
  - easy for read requests
  - on updates, data must be locked and updates propagated
  - locking strategies: lock the primary copy, a majority, all?
  - update from the clients copy or the primary copy
- performance advantage falls off fast as writes increase, because of coordination cost.

Replication

- easier for clients, since each replica contains all the functionality
  – existence of replicas can be hidden from users
- ORB (object resource broker) implementation may have a replica of the service on every workstation:
  – application accesses it locally to find the location of a service
  – ORB either forwards the request itself or returns a handle such as a network address
  – ORBs update each other
- mirrored web sites provide additional access paths to data
- update permission and propagation; overhead costs
- fault tolerance aspect (mirrored disks, redundant databases)

Replication patterns

- similar to partitioning
- capacity gain
- expect reduced communication costs by using a nearby replica
- BUT gain may be nullified by
  - implied extra functionality in the replicas, for coordination
  - network delays in the update phase

A replication principle

- Replicate a task if the cost of operation, including consistency management, is low enough to give increased capacity. Consider additional costs at all replicas, and the shift in communications costs
- natural replicas:
  - all transactions are reads, so there is no consistency management
    - – updates can be done in down time?
  - non-conflicting writes