OS for Database systems
OS intro and Process

Seehwan Yoo

Dankook University

Disclaimer: Some slides are borrowed from UC. Berkeley's 2014 OS and system programming

About Me

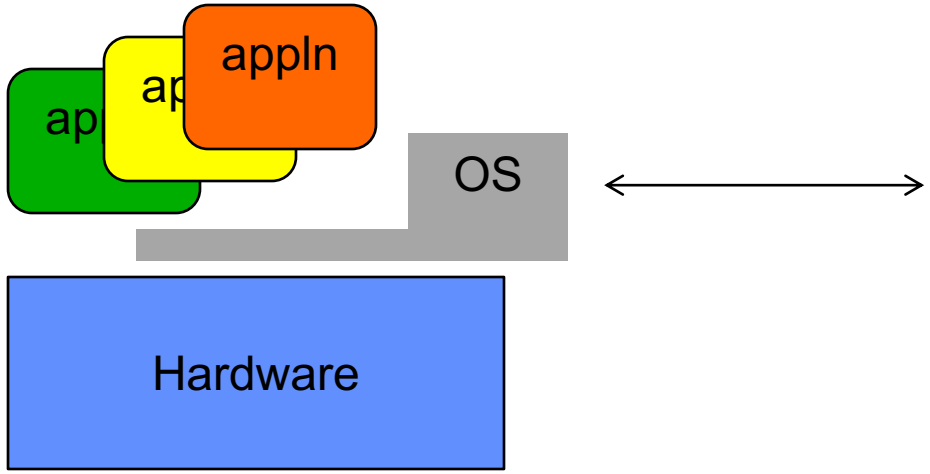
- 유시환
  - 2014~ 단국대 모바일시스템공학과 부교수
  - 2013 LG전자 CTO SWP 연구소
  - 2008 MS Research Asia research intern
- 관심분야
  - 모바일 OS, 클라우드 컴퓨팅, 시스템 보안, 시스템 가상화, 컴퓨터 시스템
  - 모바일 OS 연구실 (국제관 309호)
- 강의
  - 컴퓨터구조, OS, 인터넷/모바일프로그래밍, 클라우드 컴퓨팅
- 연락처
  - seehwan.yoo@dankook.ac.kr
  - 국제관 615호, 수업 시간, 수업 전후로 상담 가능

목차

- 강의 개요
- OS 이론 강의 (8/29~9/2)
  - OS 개론, 프로세스. + mini 실습
  - 병렬성. + mini 실습
  - 메모리, IPC. + mini 실습
  - 파일시스템, 네트워킹. + mini 실습
  - 클라우드 컴퓨팅 및 시스템 관리
- OS 실습 강의 (9/19~9/23)
  - 리눅스 개발 환경 구축 및 쉘 프로그래밍 (1d)
  - 다중쓰레드 자료구조 실습 프로젝트 (2d)
  - AWS 실습 및 WordPress 기반 DB 실습 프로젝트 (2d)

What is an operating system?

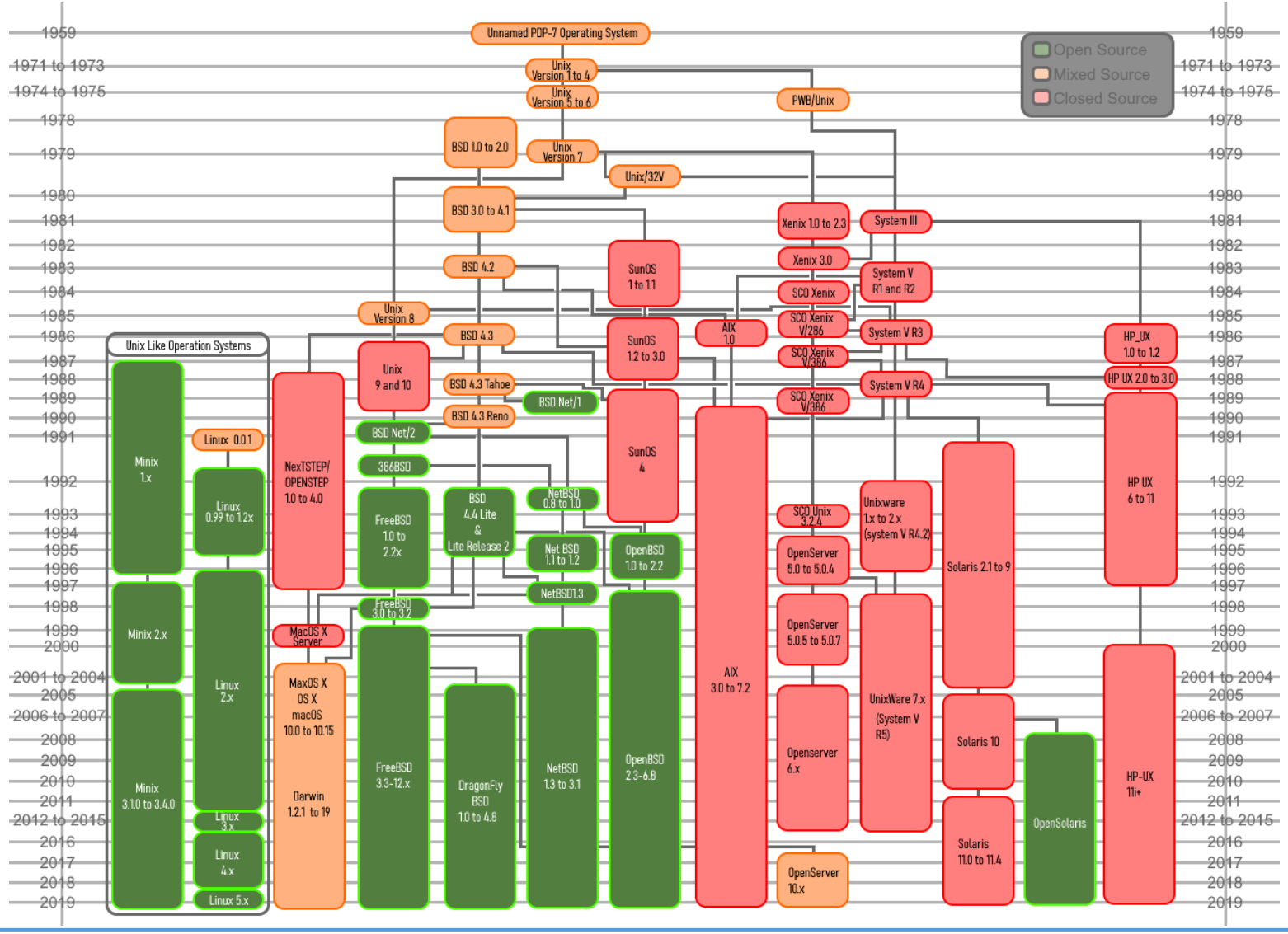
- Special layer of software that provides application software access to hardware resources
  - Convenient abstraction of complex hardware devices
  - Protected access to shared resources
  - Security and authentication
  - Communication amongst logical entities



What is an Operating System?

- Referee
  - Manage sharing of resources, Protection, Isolation
    - Resource allocation, isolation, communication
- Illusionist
  - Provide clean, easy to use abstractions of physical resources
    - Infinite memory, dedicated machine
    - Higher level objects: files, users, messages
    - Masking limitations, virtualization
- Glue
  - Common services
    - Storage, Window system, Networking
    - Sharing, Authorization
    - Look and feel

UNIX history https://en.wikipedia.org/wiki/History_of_Unix



Unix intro

- https://www.slideserve.com/mireya/introduction-to-unix-powerpoint-ppt-presentation
- For multiple users
- For multiple programs execution
- UNIX cf.) MULTICS
  - Multics focused on multi-user; but the hardware were not powerful to properly support it
  - as a counterpart, Unix (uni-x) was introduced
- Linux and UNIX?
  - Linux started as a free x86 variant

OS: a human-computer interface

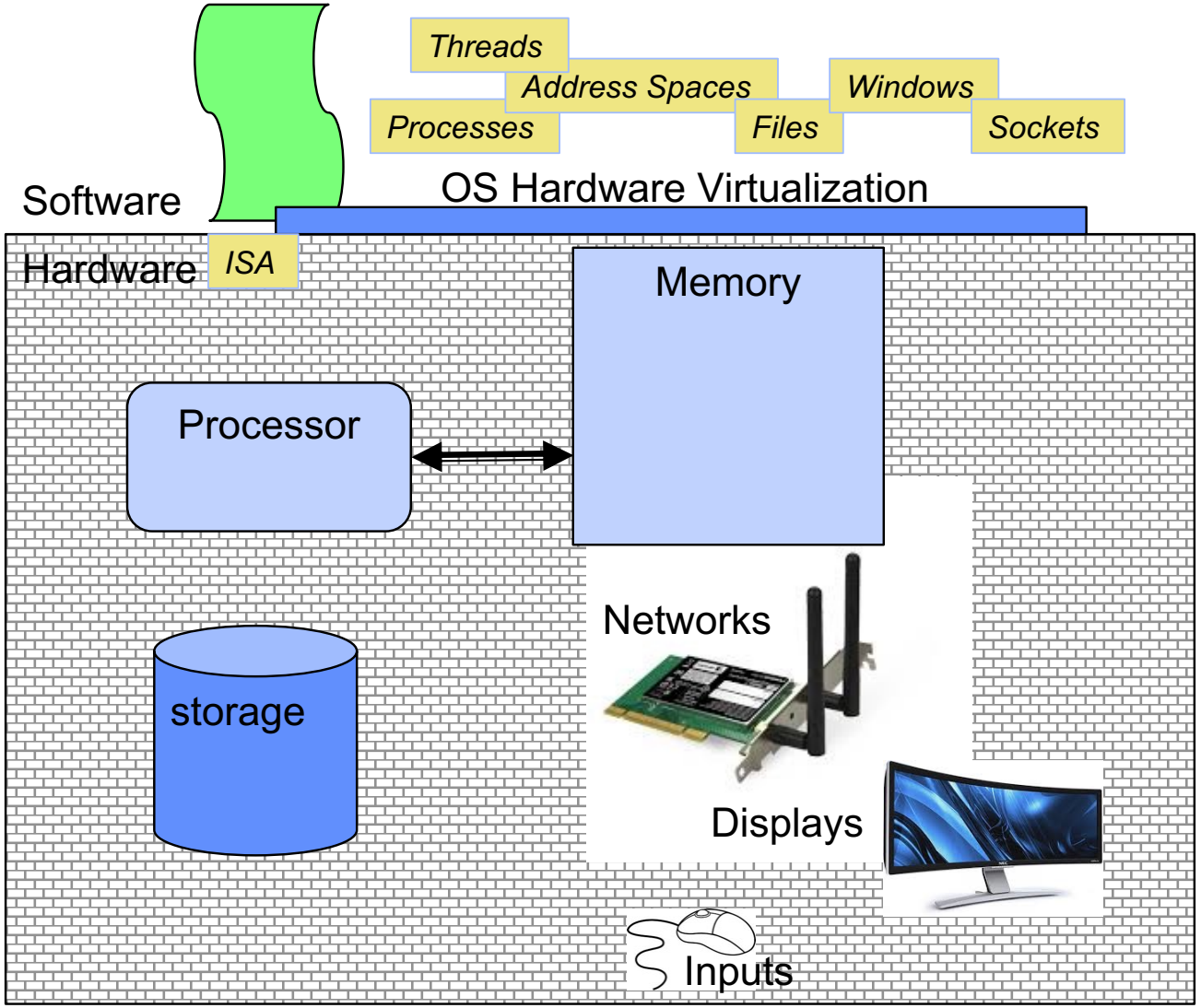
- defines how we interact/use with computers
- take some input from user, run as requested
  - called ‘shell’
- Program into process
  - stored entity into running instance
- to share CPU, memory and I/O devices among multiple processes
  - we use ‘abstractions’
  - logical things, concepts
  - For example, independent data unit - file
    - where you’re storing your data, and how we can call/identify it
    - i-nodes, directories

OS, a practical perspective

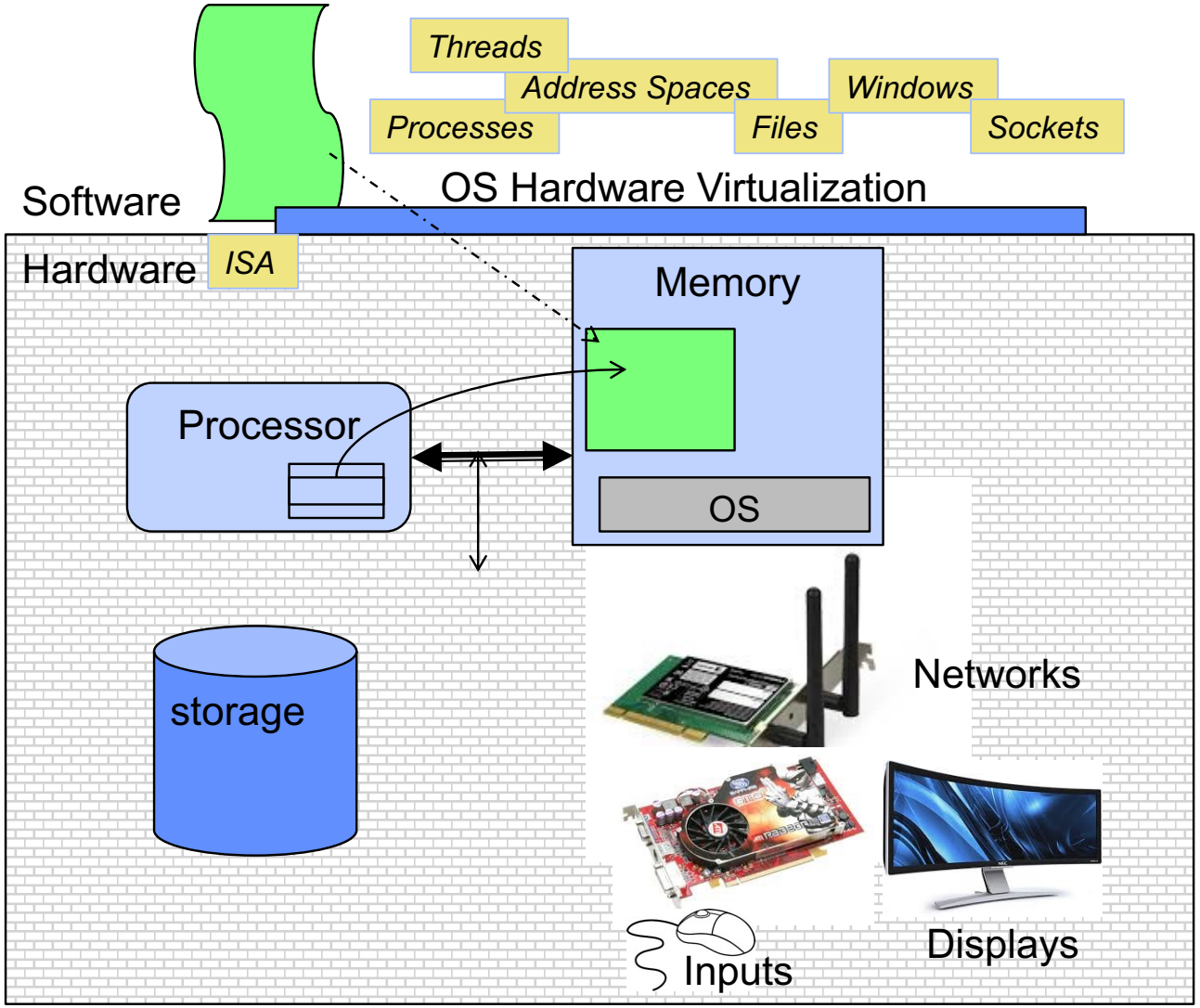
- a tool and a way use to computer
- + a full suite of software tools to operate it
- e.g.) file read, file write, change directory, create/delete directory, taking user input, loading library, executing program, compiler, editor
  finding patterns, etc.
- find something in /usr/bin /bin directory
- including GUI and graphical libraries
- They are all part of OS, and we will talk a part of it

OS: big picture

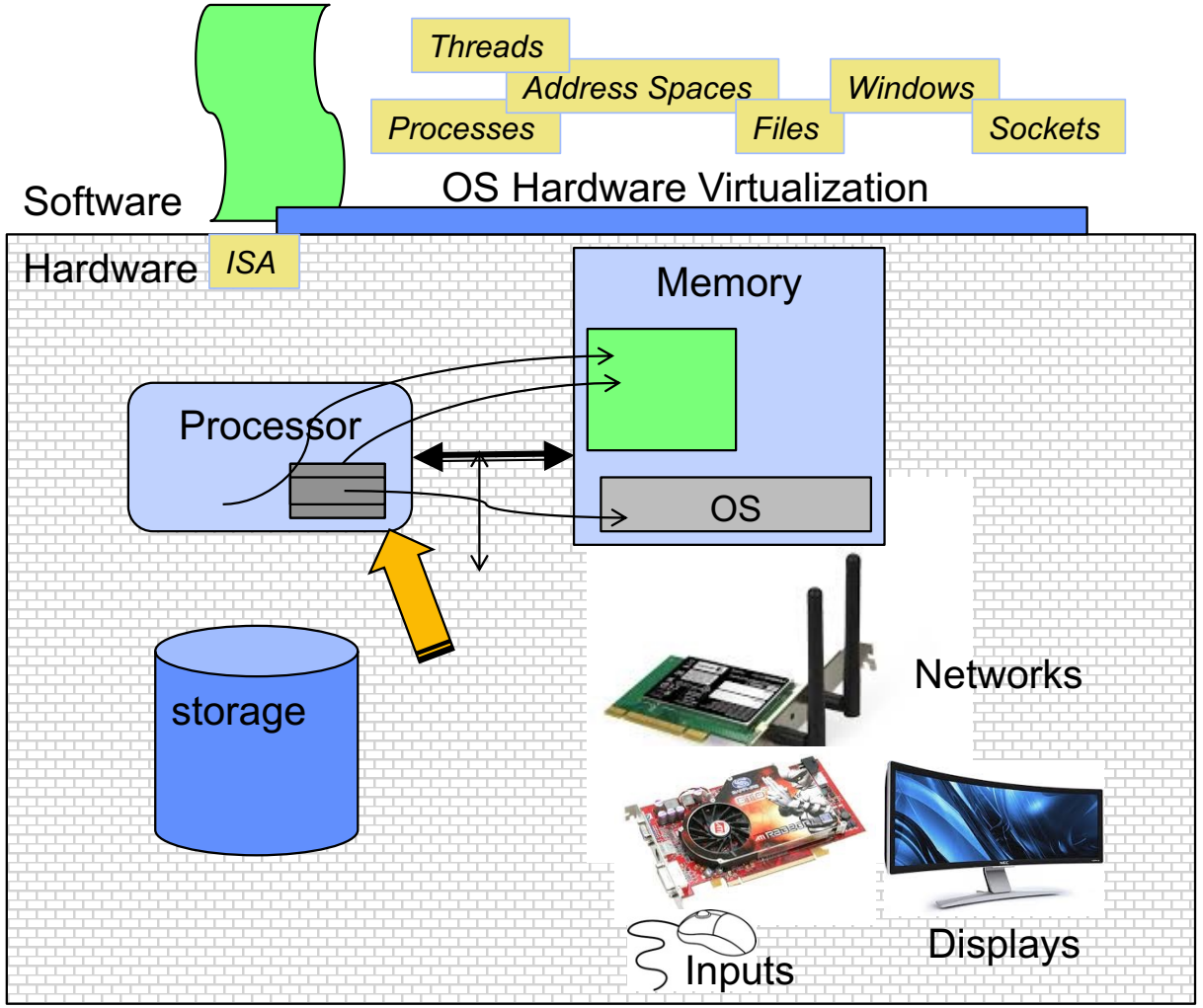
OS Basics: "Virtual Machine" Boundary



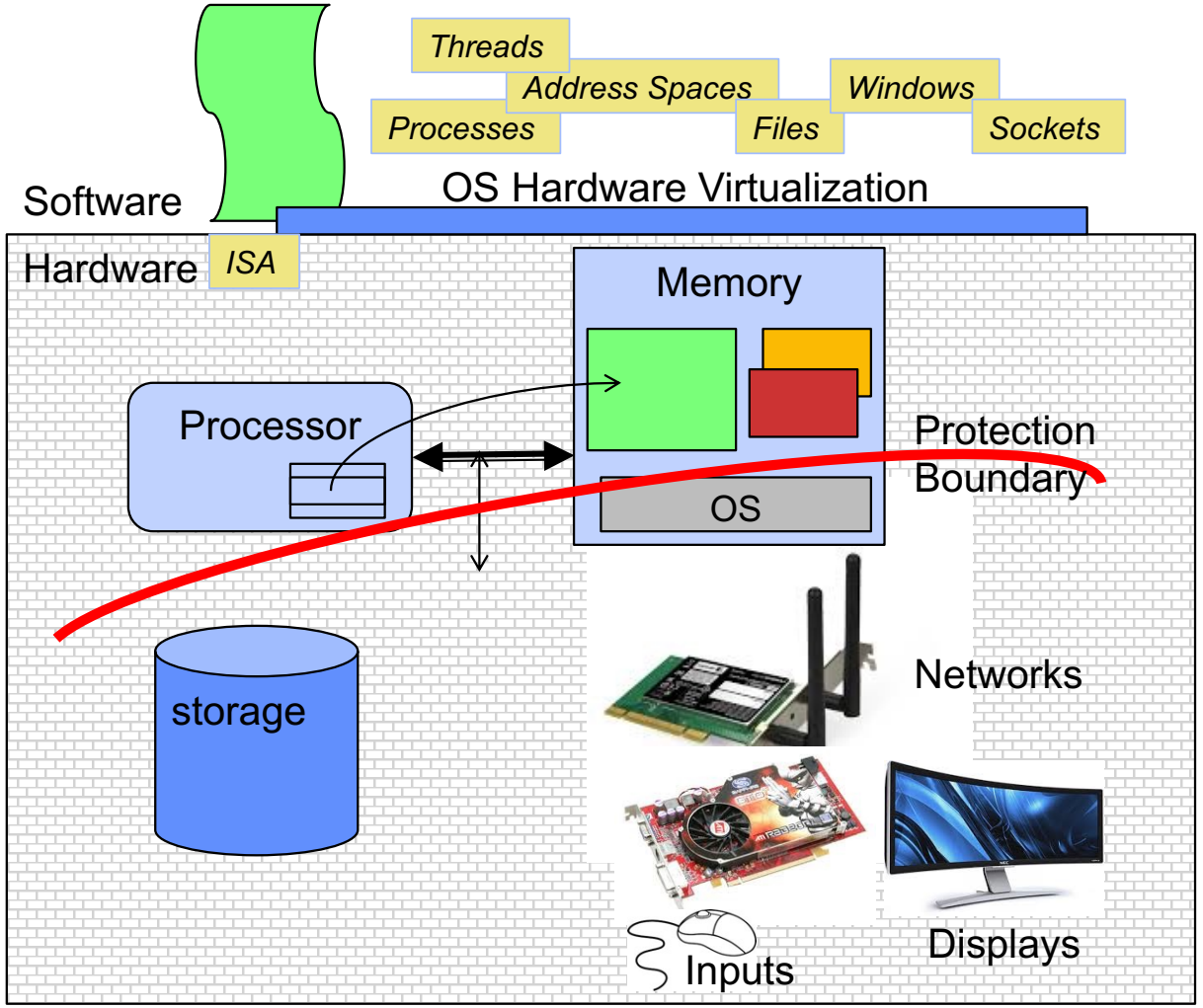
OS Basics: Program => Process



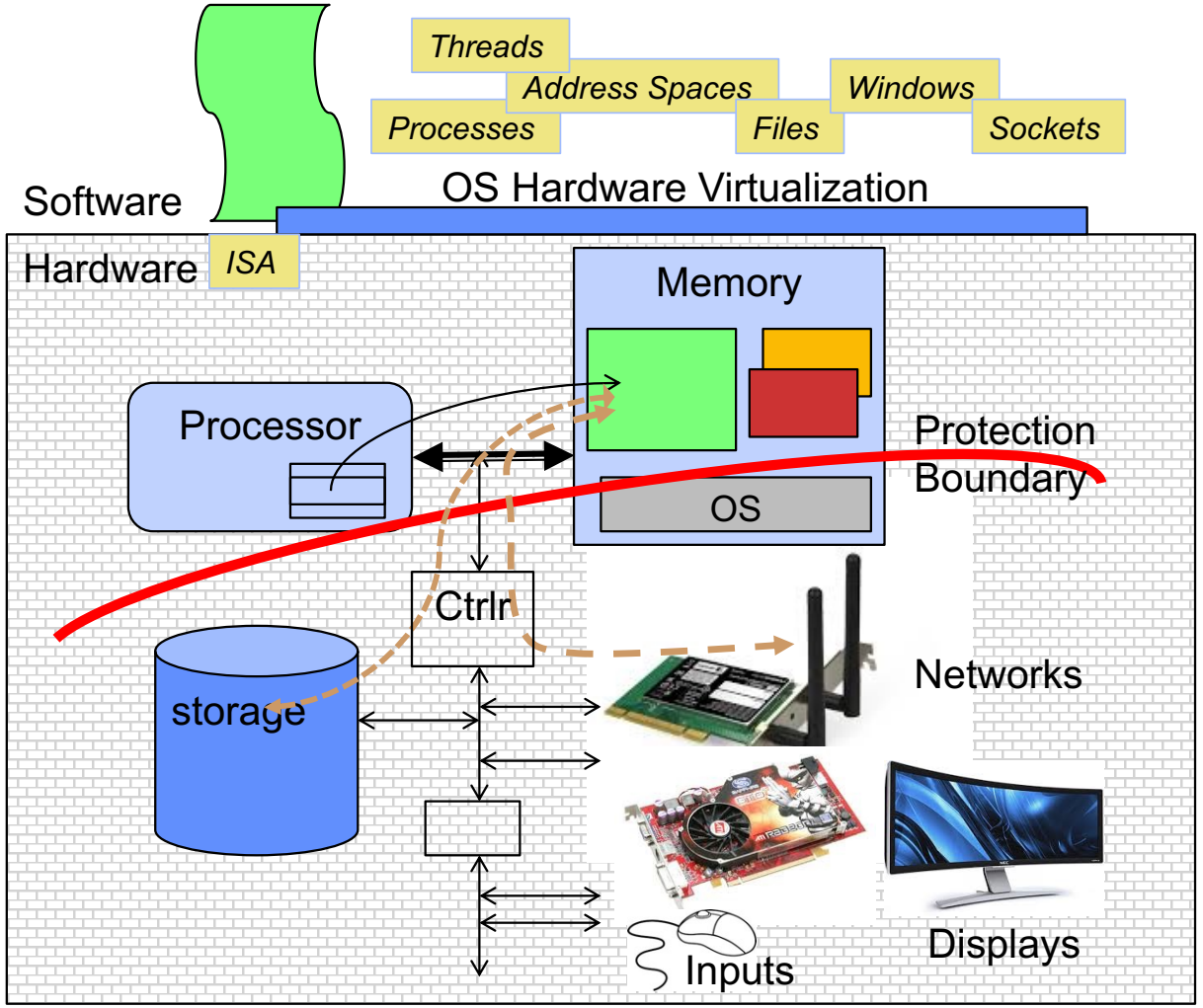
OS Basics: Context Switch



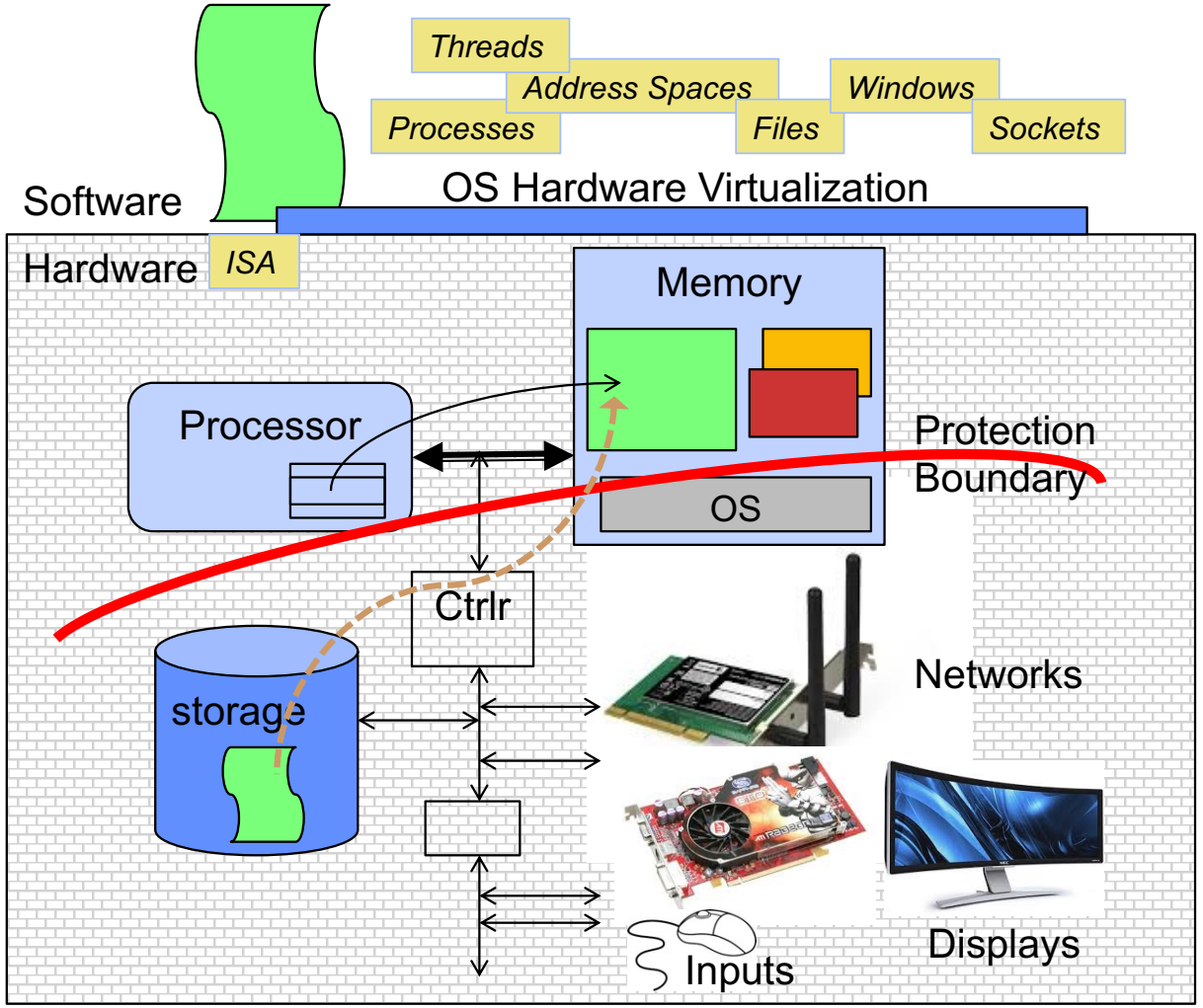
OS Basics: Scheduling, Protection



OS Basics: I/O


Threads
Address Spaces
Windows
Processes
Files
Sockets
Software
OS Hardware Virtualization
Hardware
ISA
Memory
Processor
Protection
Boundary
OS
Ctrlr
storage
Networks
Inputs
Displays

OS Basics: Loading

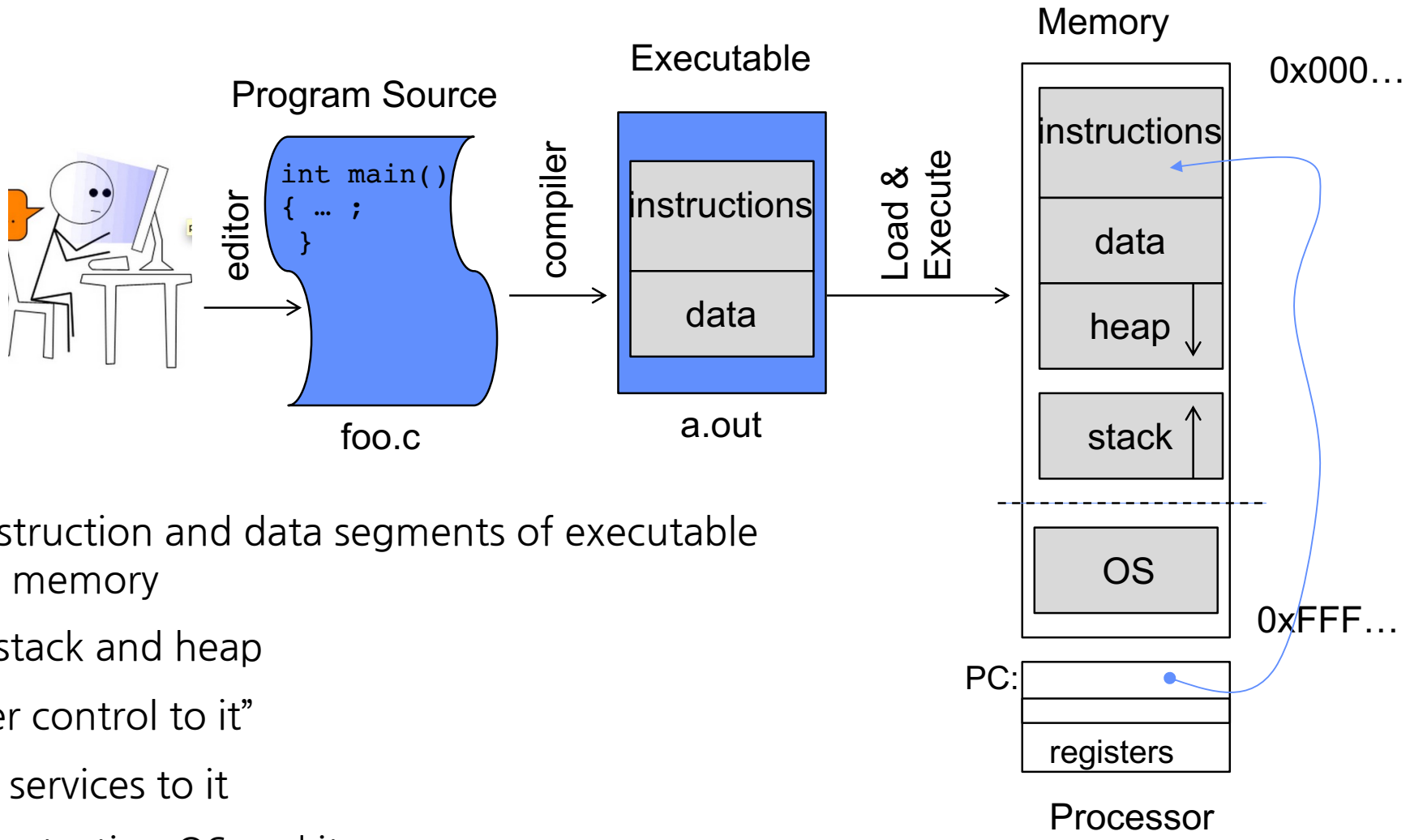


Introduction to the process

Four fundamental OS concepts

- Process
  - An instance of an executing program is *a process consisting of an address space and one or more threads of control*
- Privileged/User Mode (Dual mode operation)
  - The hardware can operate in two modes, with only the “system” mode having the ability to access certain resources.
- Protection
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses
- Address Space
  - Programs execute in an *address space* that is distinct from the memory space of the physical machine

OS Bottom Line: Run Programs



- Load instruction and data segments of executable file into memory
- Create stack and heap
- “Transfer control to it”
- Provide services to it
- While protecting OS and it

Looking into executable: objdump

--- add_some_number.c --- in the git OS_hw0

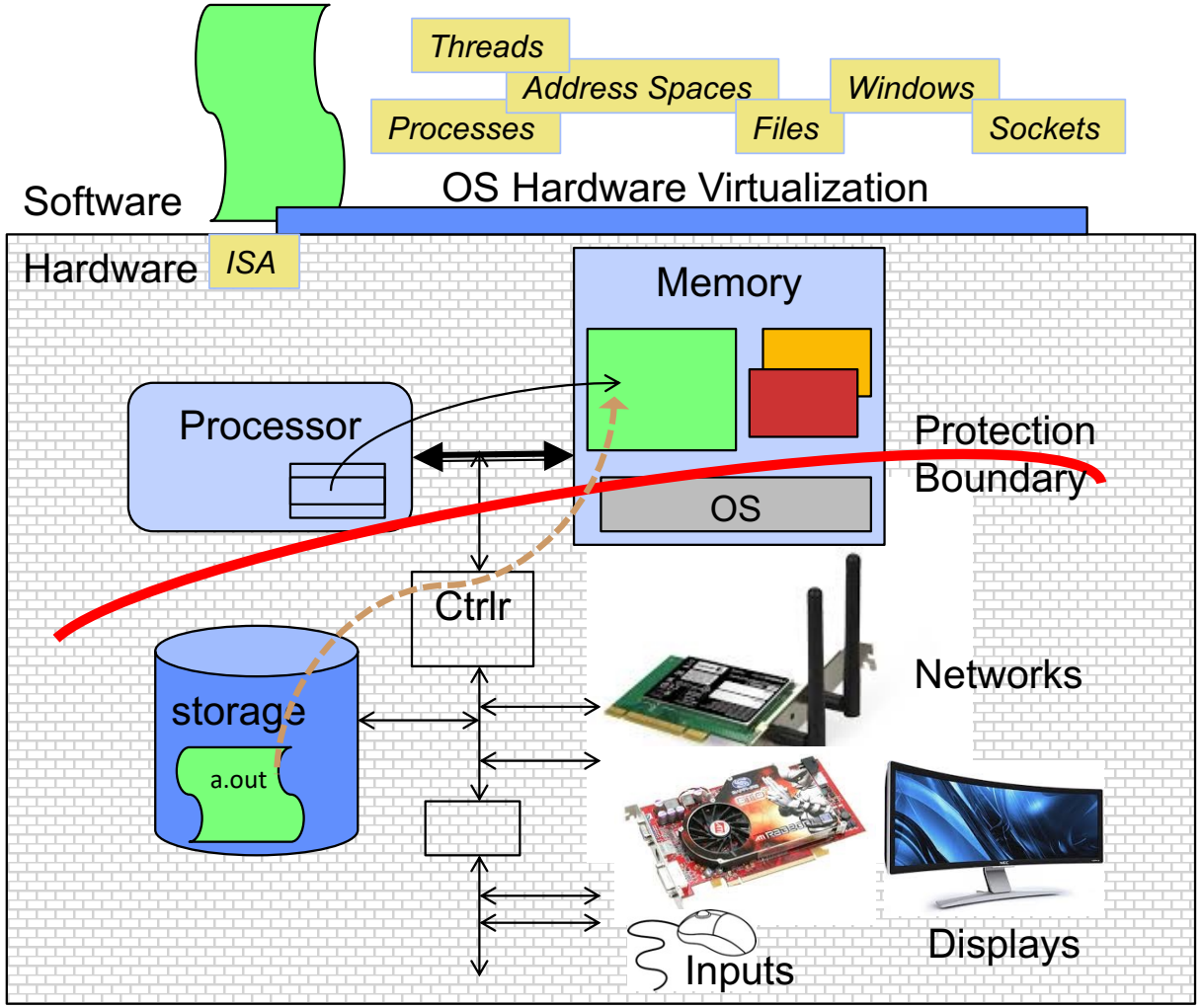
```
int sum = 0;
int main(int argc, char * argv[])
{
  int number = 0;
  int i = 0;
  if (argc == 2) {
    number = atoi(argv[1]);
    if ((errno == ERANGE &&
        (number == LONG_MAX ||  number == LONG_MIN))
        || (errno != 0 && number == 0)) {
            perror("strtol");
            exit(EXIT_FAILURE);
    }
  } else  {
    number = 100;
  }

  for (i = 0 ; i <= number ; i++) {
    sum += i;
  }
  printf("sum from 0 to %d: %d (0x%x)\n",
            number, sum, sum);
  return 0;
}
```

compile the code, assuming that outputfile is a.out
>file a.out
>readelf --help
>readelf -h a.out
>readelf -h -S -l a.out

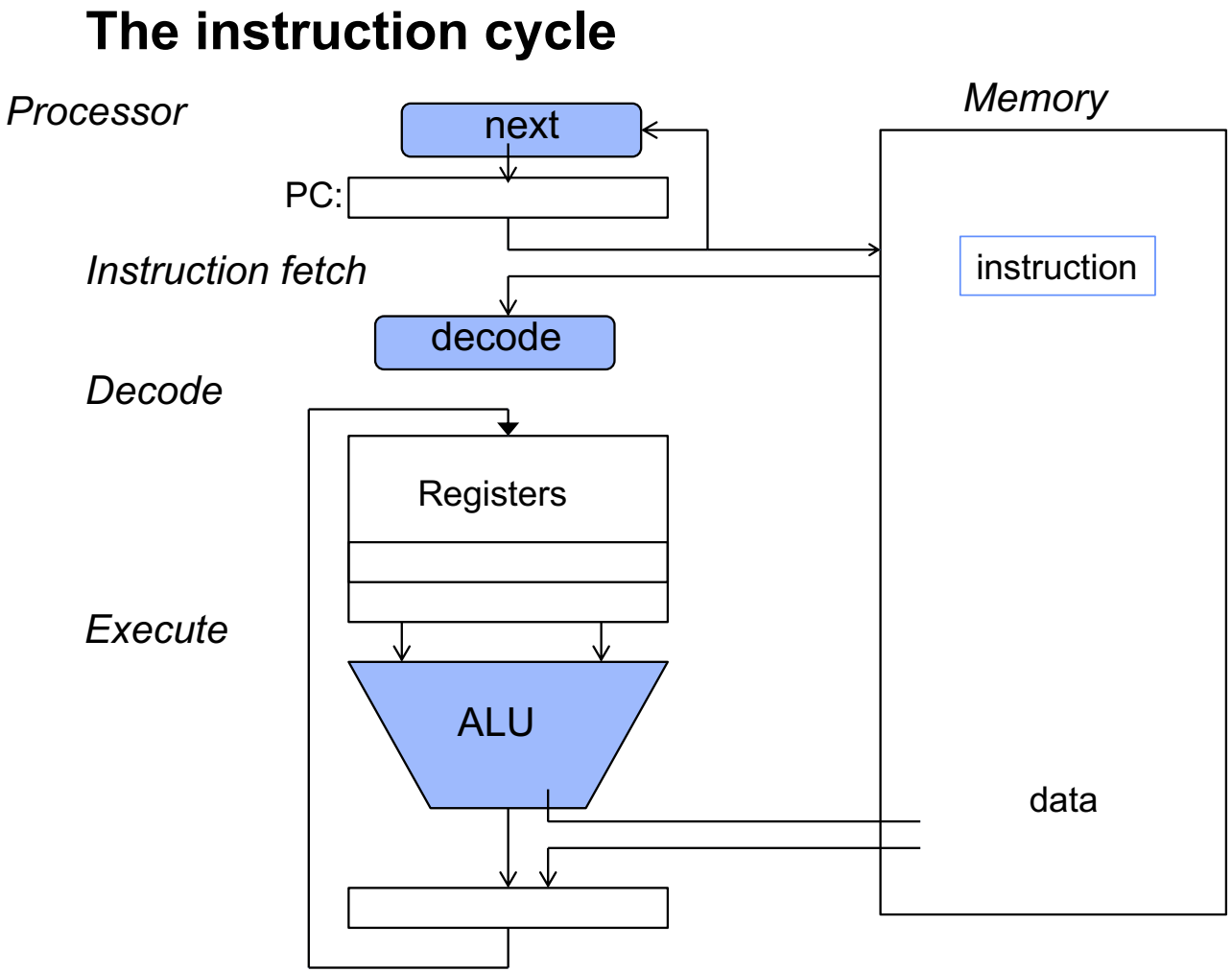
>objdump --help
>objdump -x a.out
…
.text main
>objdump -d a.out
code

>objdump -x a.out
…
.data
…
>nm -x a.out
data

OS basics: Loading



revisit: execution of a program inside CPU



break question

- Why no ‘stack’ or ‘heap’ in executable?

Key OS Concept: the Process

***Process* = Execution of a Program with Restricted Rights**

"User Programs"

"Applications"

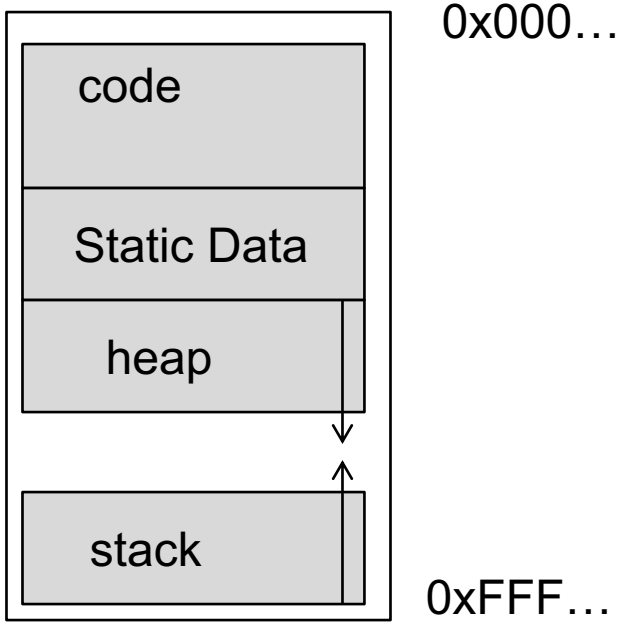
---------- Operating System Boundary -------

***Process* = Address Space with one or more threads of control**

"Kernel"

"Operating System"

Memory Layout of a process



- What's in the code segment? Data segment?
- What's in the stack segment?
  - How is it allocated?
  - How big is it?
- What's in the heap segment?
  - How is it allocated?
  - How big?

Looking into Stack pointer

- Compile with -g option
  - >gcc -g add_some_number.c -o add_some_number
- run GDB with add_some_number
  - >gdb ./add_some_number
- set breakpoint at main
  - (gdb) b main
- run the process
  - (gdb) run
- It will stop at breakpoint, which is set at main() routine

- check register values
  - (gdb) info register
  - rsp value?
  - (gdb) info stack
  - (gdb) backtrace
    (gdb) bt
  - (gdb) frame
- Process information
  - (gdb) info proc

Function calls: stacked variables & procedures


```
--- fibonacci.c --- in the git OS_hw0
int main(int argc, char * argv[])
{
...
  sum = fib(number);
  printf("%d-th Fibonacci number: %d (0x%x)\n",
number, sum, sum);
  return 0;
}


int fib(int number)
{
  if (number < 3) return 1;
  else return (fib(number-1)
         + fib(number-2));
}
```

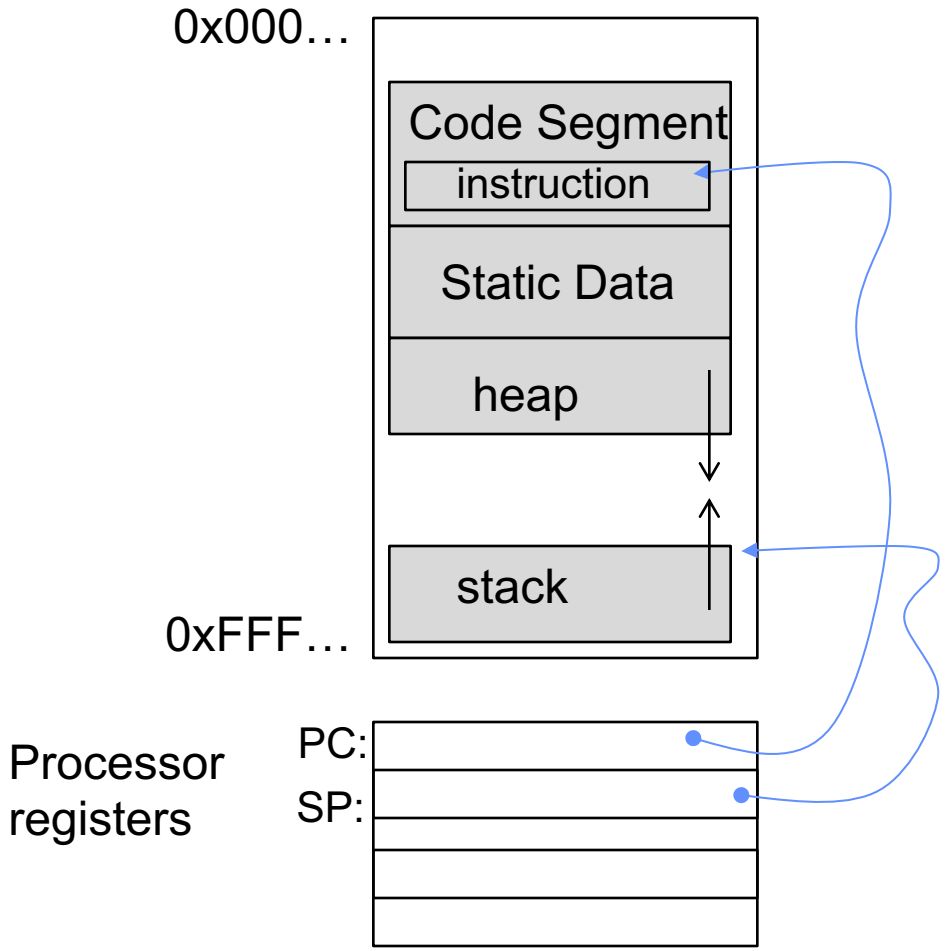
- compile with –g option

- run gdb with fibonacci
- set breakpoint at fib function
  - (gdb) b fib
- run program with argument 4
  - (gdb) run 4
- It will stop at the fib() function
  - (gdb) print number
- Check stackpointer
  - (gdb) info register
- backtrace stack frames
  - (gdb) bt
  - You're at fib()
    from main()
- continue execution (until the next breakpoint)
  - (gdb) continue
- It will stop at fib() function
  - (gdb) print number
- Check stackpointer
  - (gdb) info register
- backtrace stack frames
  - (gdb) bt

Thread of Control

- Thread
  - Unit of execution (on a processor)
- Usually a program has at least one thread
  - A thread is executing on a processor when it is resident in the processor registers.
  - A program is running
- *Thread context*
  - Registers hold the root state of the thread
  - general-purpose registers
  - stack pointer
  - program counter
  - May be defined by the instruction set architecture or by compiler conventions
  - The rest is “in memory”

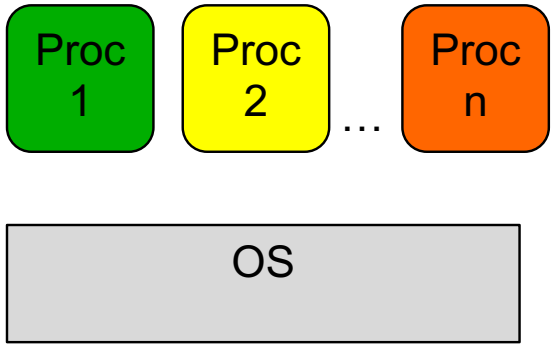
In a Picture

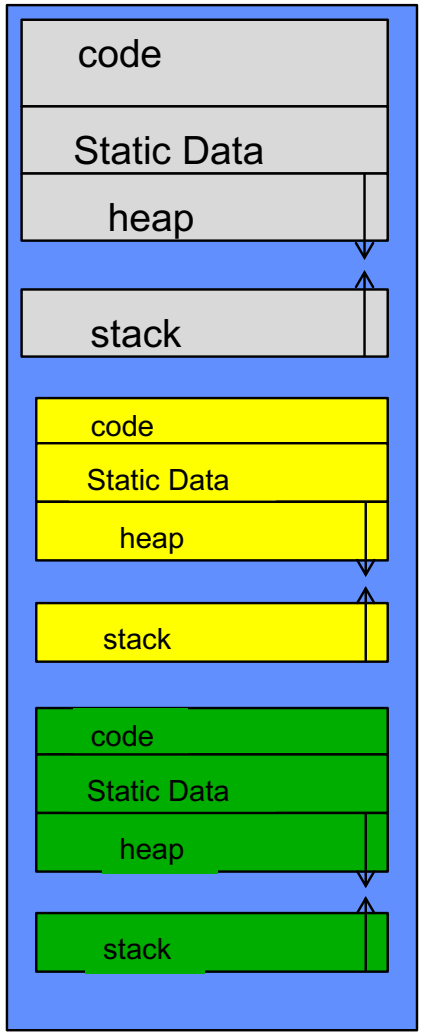

0x000…
Code Segment
instruction
Static Data
heap
stack
0xFFF…
Processor registers
PC:
SP:

Multiple threads (multithreads)

- Can be possible!
- Multiple execution units even in a program
  - multiple PC
  - multiple stack
  - multiple threads context
  - Multi-core, multi CPU, GPU processing
- Thread-level parallelism
- OS support is essential
- Will be addressed in later classes

Multiprogramming - Multiple Processes





Key OS Concept: Protection

- Protection target
  - Operating System itself from user programs
    - Reliability: compromising the operating system generally causes it to crash
    - Security: limit the scope of what processes can do
    - Privacy: limit each process to the data it is permitted to access
    - Fairness: each should be limited to its appropriate share
  - User programs from one another
- Mechanisms
  - Primary Mechanism: Dual mode operation, Address space
    - Limit the accessible (executable) memory region and address
    - Can only touch what is mapped in
  - Additional Mechanisms
    - Privileged instructions, in/out instructions, special registers
    - syscall processing, subsystem implementation (e.g., file access right)

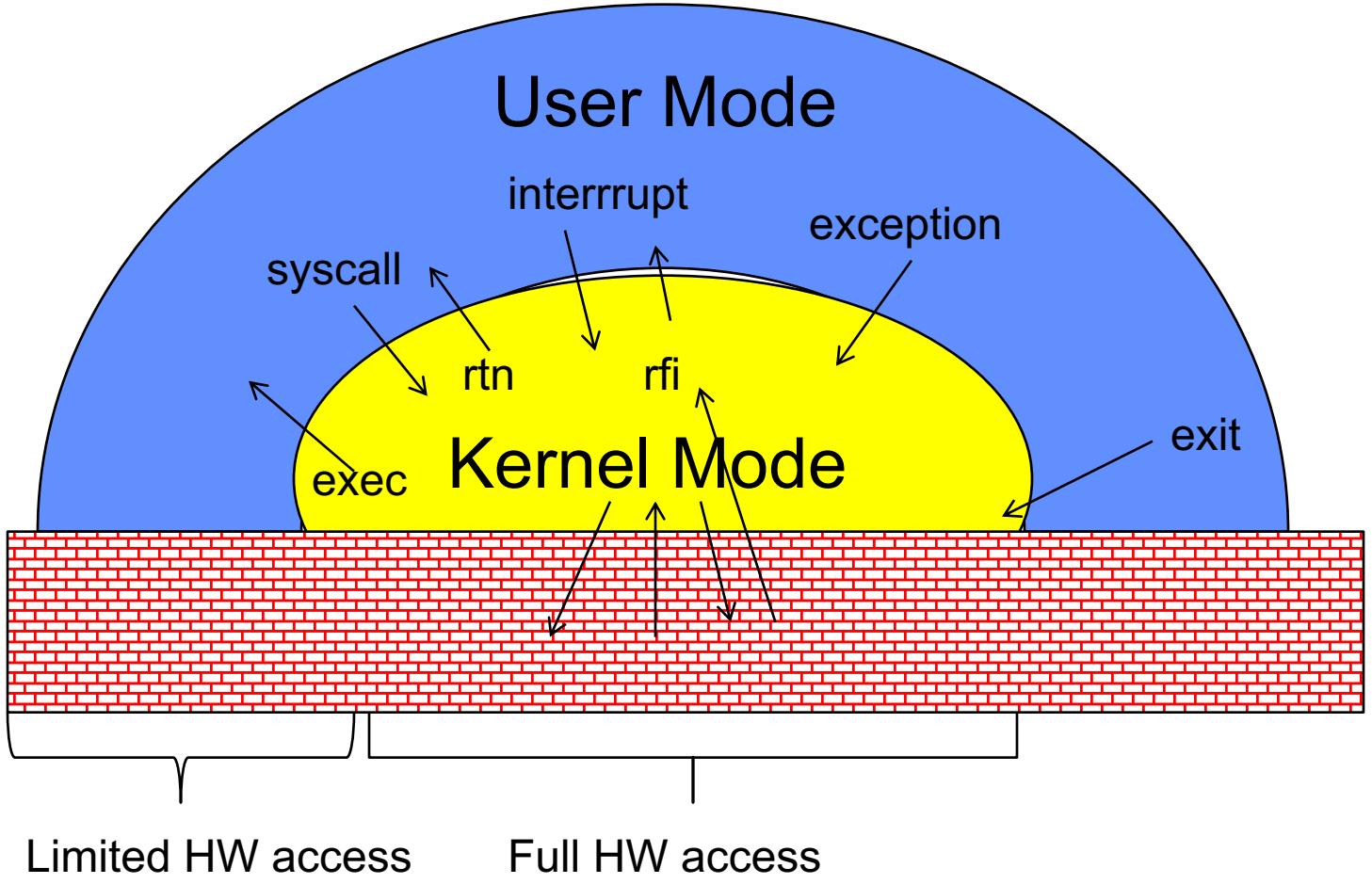
Key OS Concept: Dual Mode Operation

- Protect OS from the other software
  - OS performs administrative operations
    - set up security policy of process
    - define address space of process
    - resource sharing policy (memory, I/O)
  - if compromised, the entire system's security could be breached
- Architecture support for OS
  - modify CPU, ISA
  - CPU distinguishes OS from normal user program execution

Key OS Concept: Dual Mode Operation

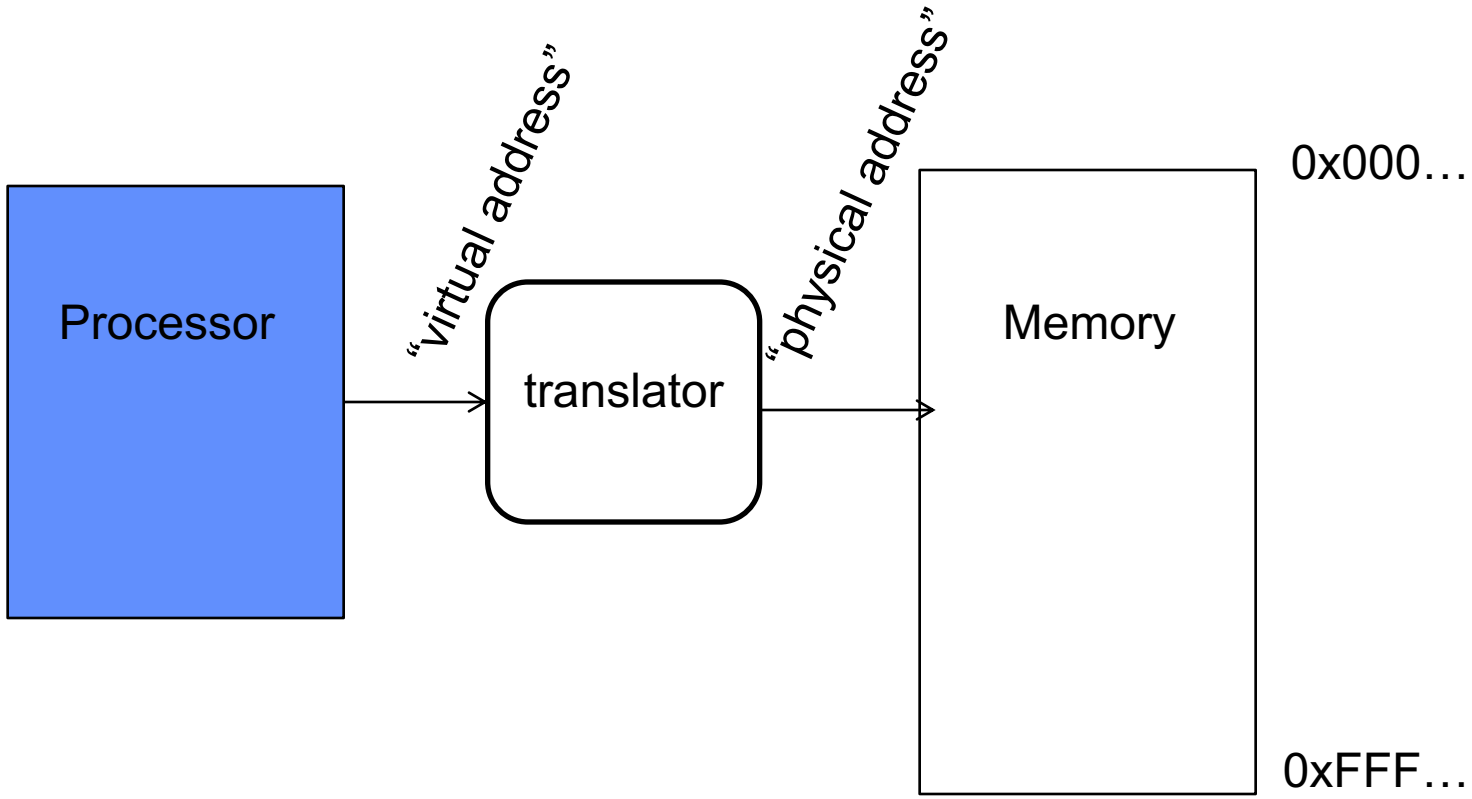
- What is needed in the hardware to support “dual mode” operatio n?
  - a bit of state (user/system mode bit)
  - Certain operations / actions only permitted in system/kernel mode
    - In user mode they fail or trap
- Mode switching
  - User->Kernel transition *sets* system mode AND saves the user PC
    - Operating system code carefully puts aside user state then performs the necessary ope rations
  - Kernel->User transition clears system mode AND restores appropriate user PC
    - return-from-interrupt
  - Only available at explicit ‘gates’

User/Kernel (privileged) Mode

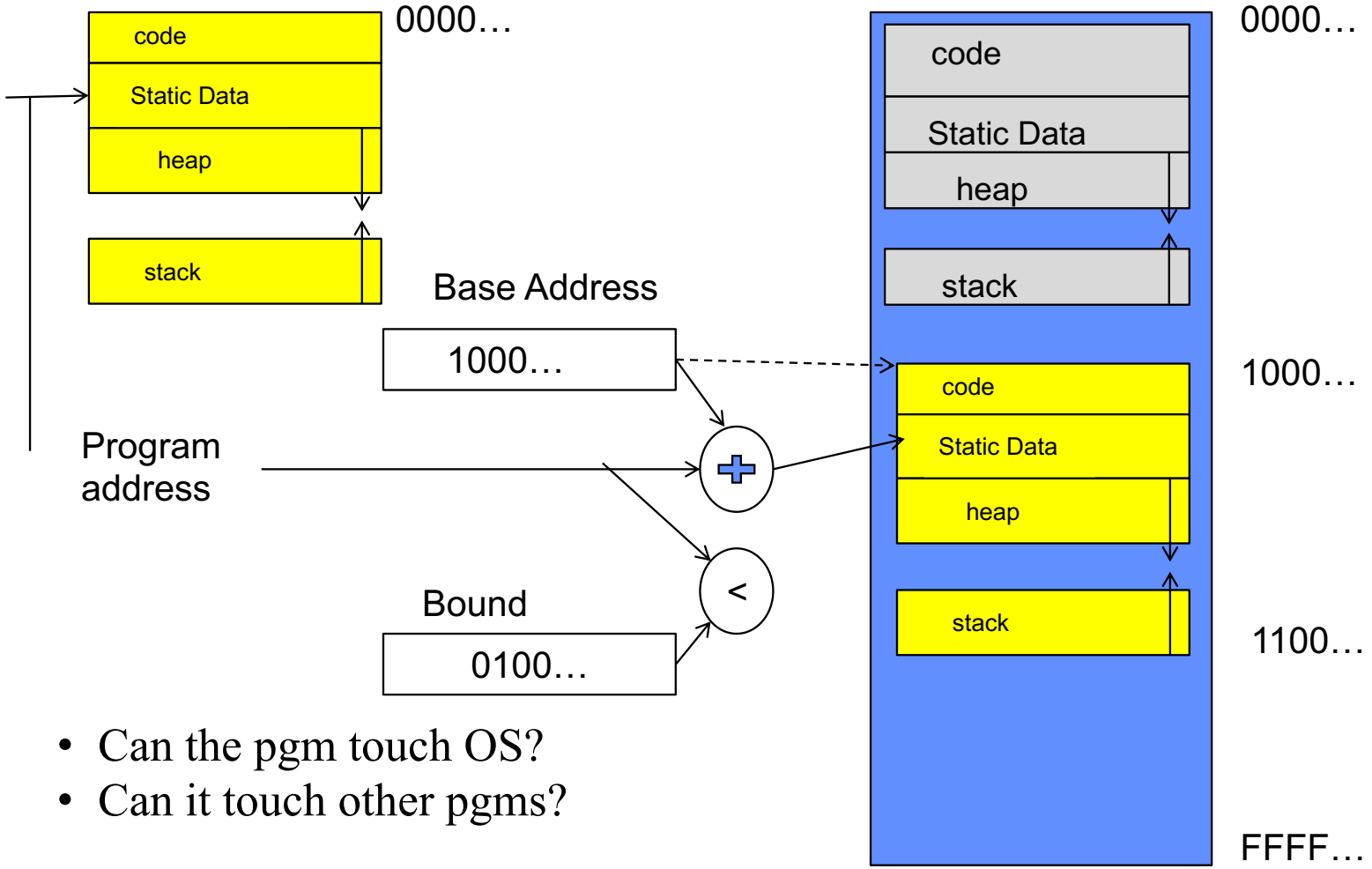


Key OS Concept: Address Space

- Program operates in an address space that is distinct from the physical memory space of the machine

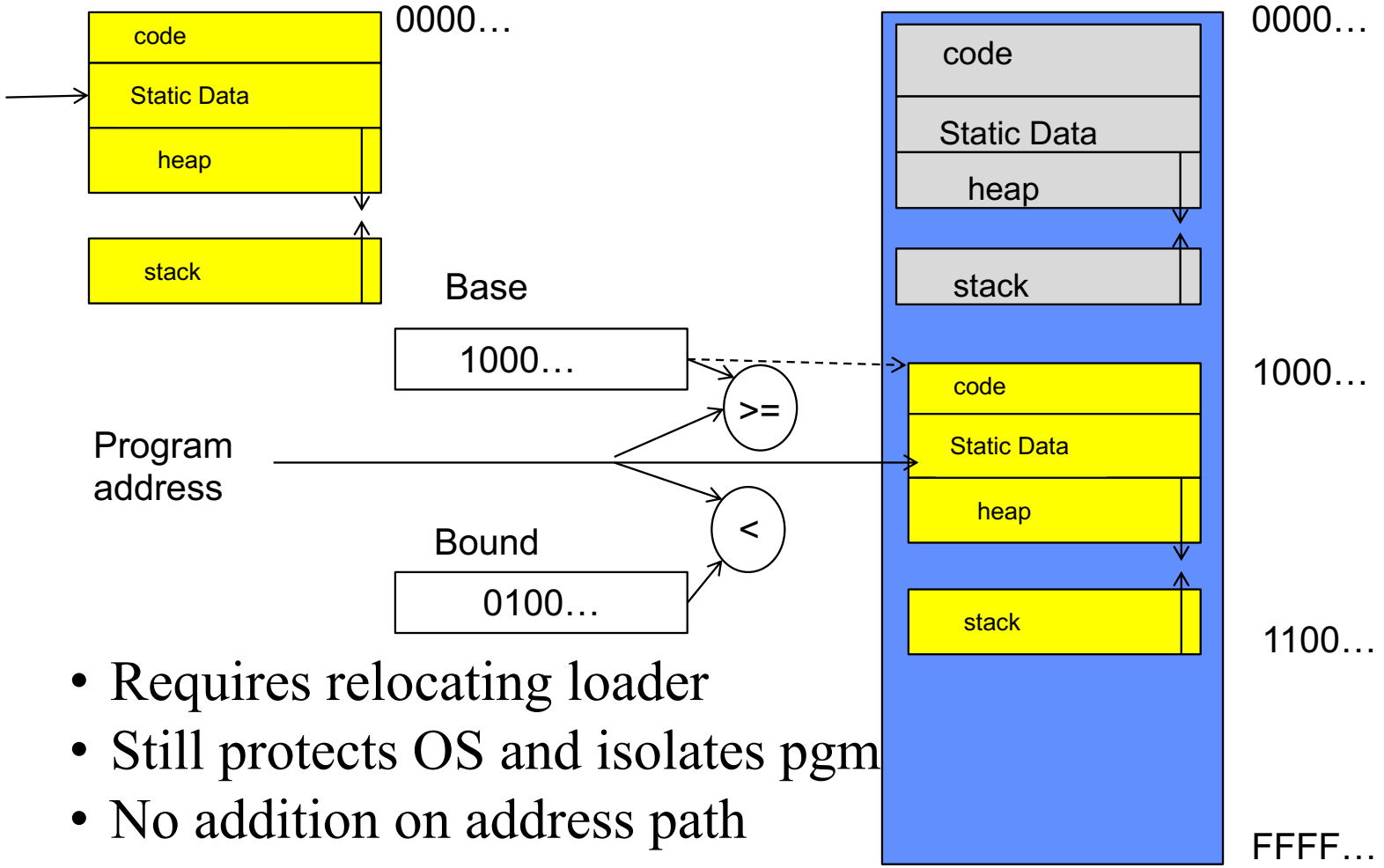


A simple address translation: B&B



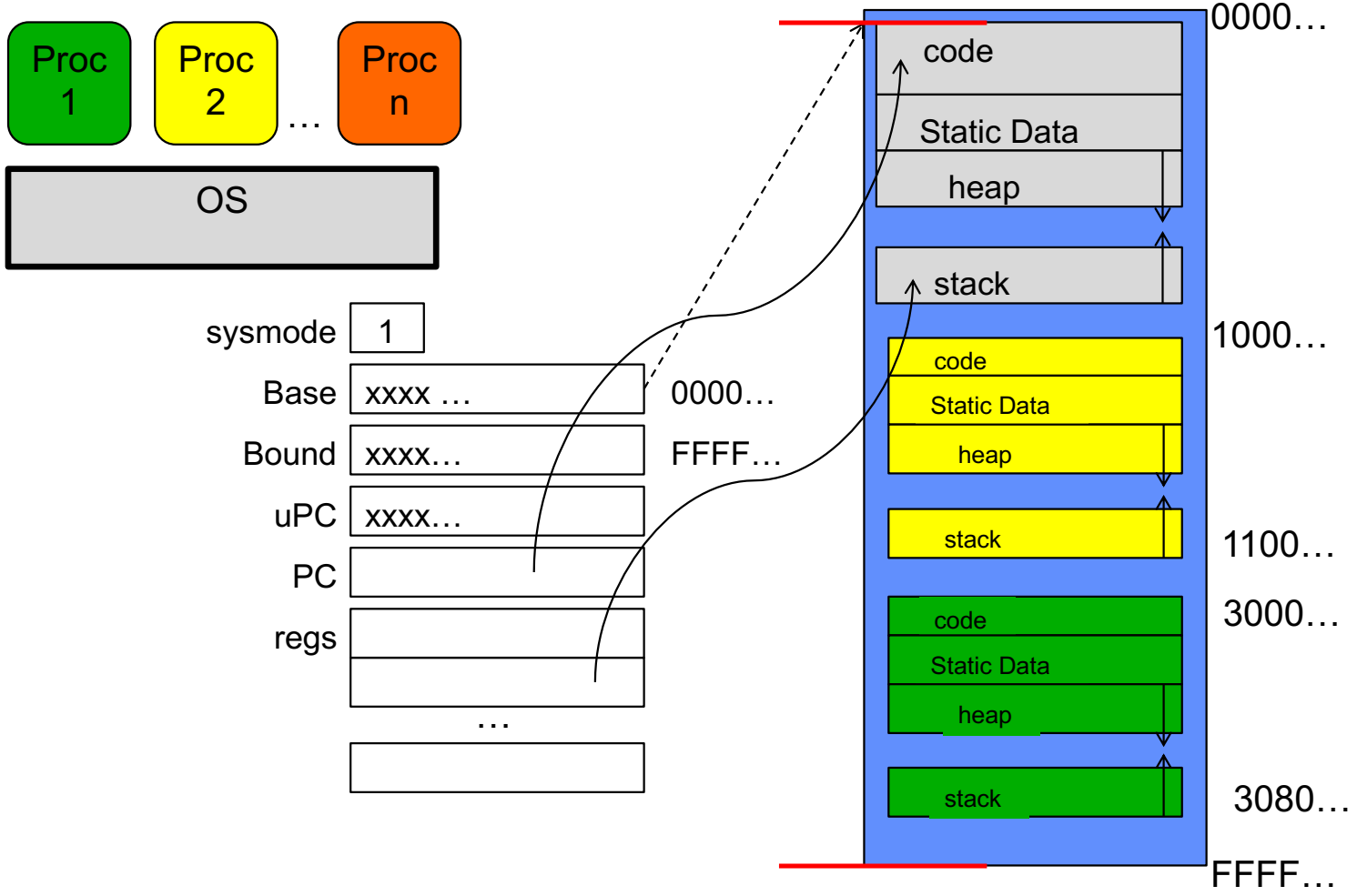
- Can the pgm touch OS?
- Can it touch other pgms?

A different base and bound

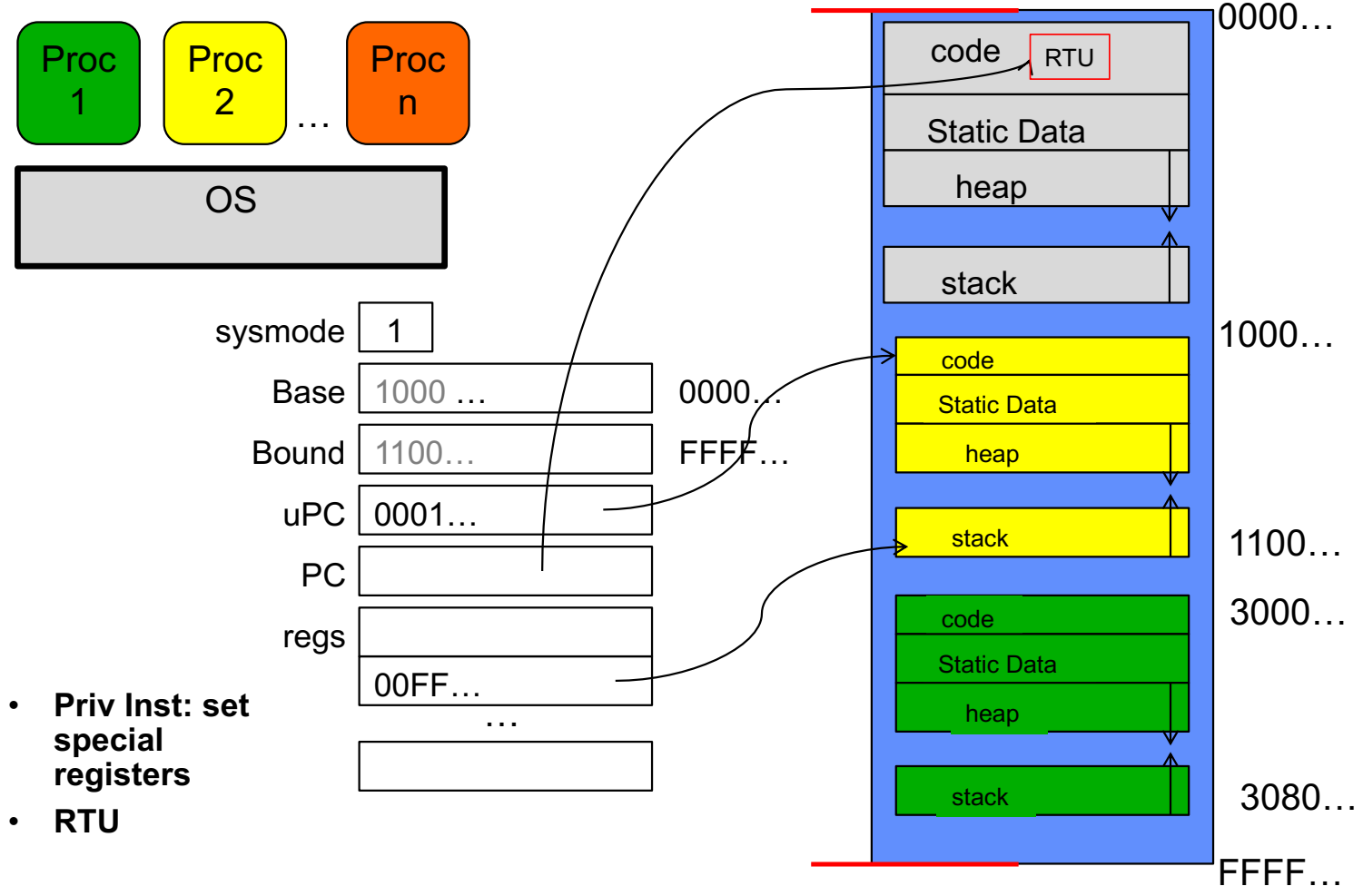


- Requires relocating loader
- Still protects OS and isolates pgm
- No addition on address path

Simple B&B: OS loads process

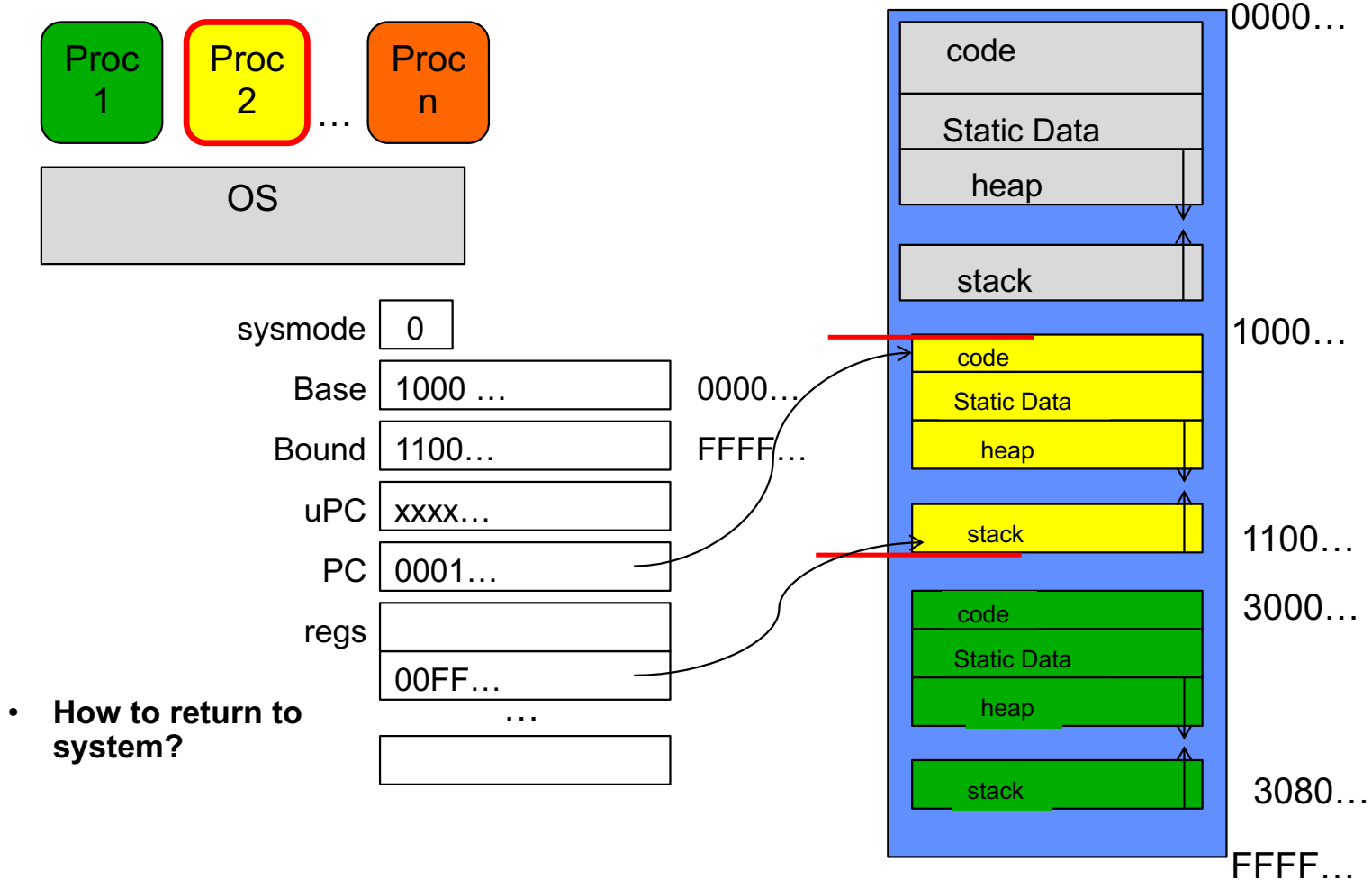


Simple B&B: OS gets ready to switch



- **Priv Inst: set special registers**
- **RTU**

Simple B&B: "Return" to User



- **How to return to system?**

Study ELF

- Investigate how OS starts a program
  - ELF
    - Executable file format
    - Executable file: file that stores a program that can be executed on a computer
  - Section
  - Relocation
  - Position Independent Code
  - How OS prepares for executing a program in ELF file
  - By next week

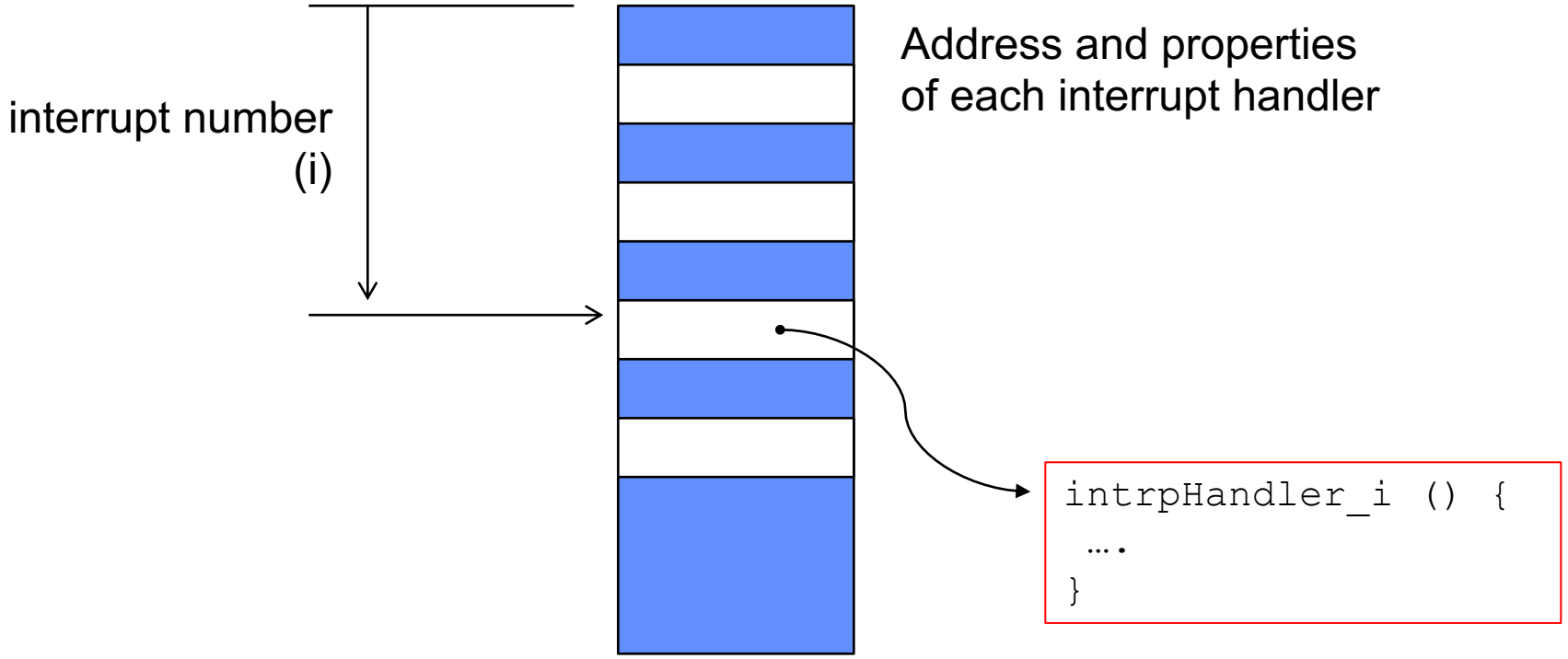
3 types of Mode Transfer

- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but “outside” the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) - for later
  - Marshall the syscall id and args in registers and exec syscall
- Interrupt
  - External asynchronous event triggers context switch
  - eg. Timer, I/O device
  - Independent of user process
- Trap or Exception
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …
- All 3 are an UNPROGRAMMED CONTROL TRANSFER
  - Where does it go?

Question

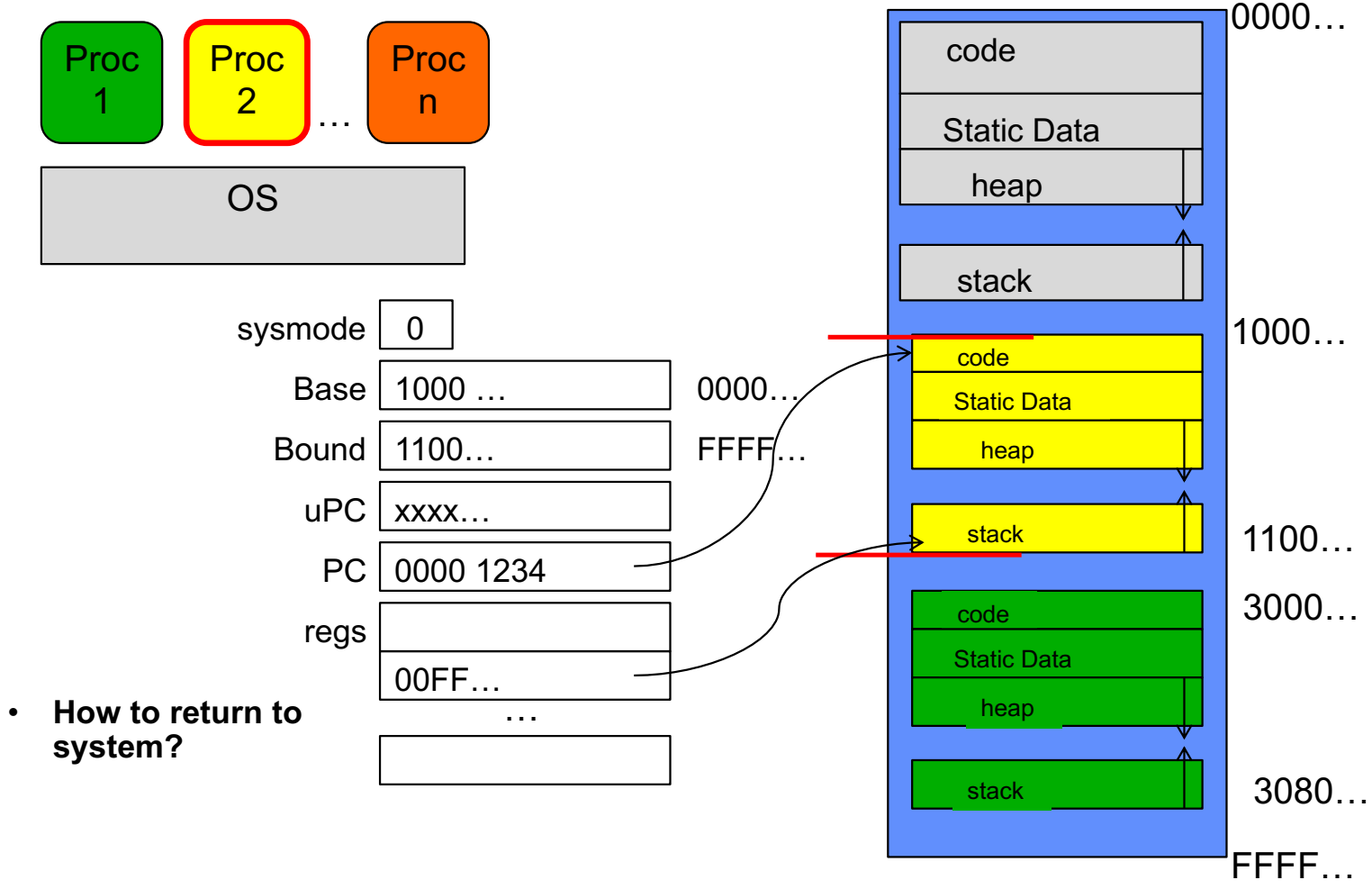
- How do we get the system target address of the “unprogrammed control transfer?”

Interrupt Vector



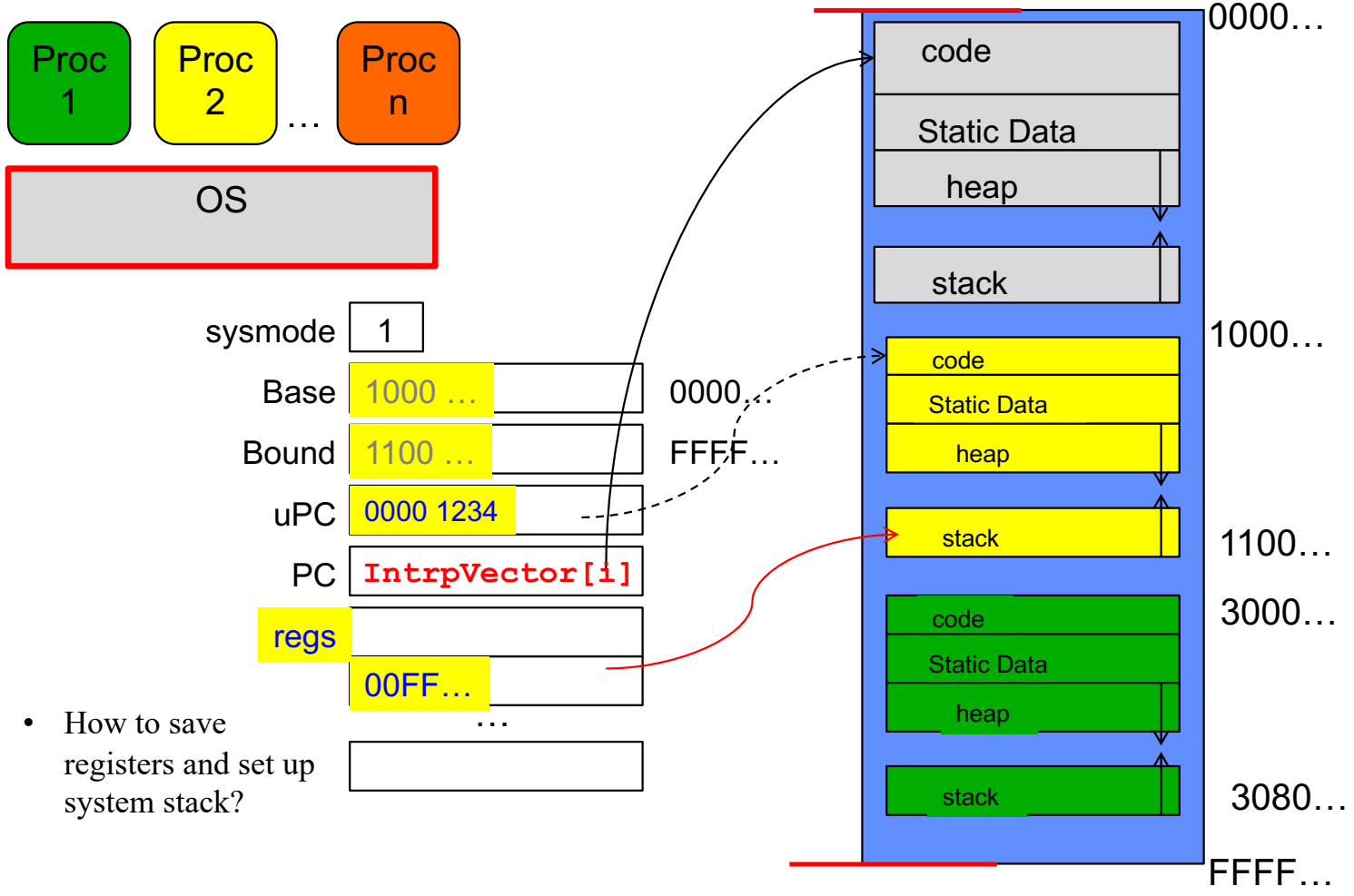
- Where else do you see this dispatch pattern?

Simple B&B: User => Kernel



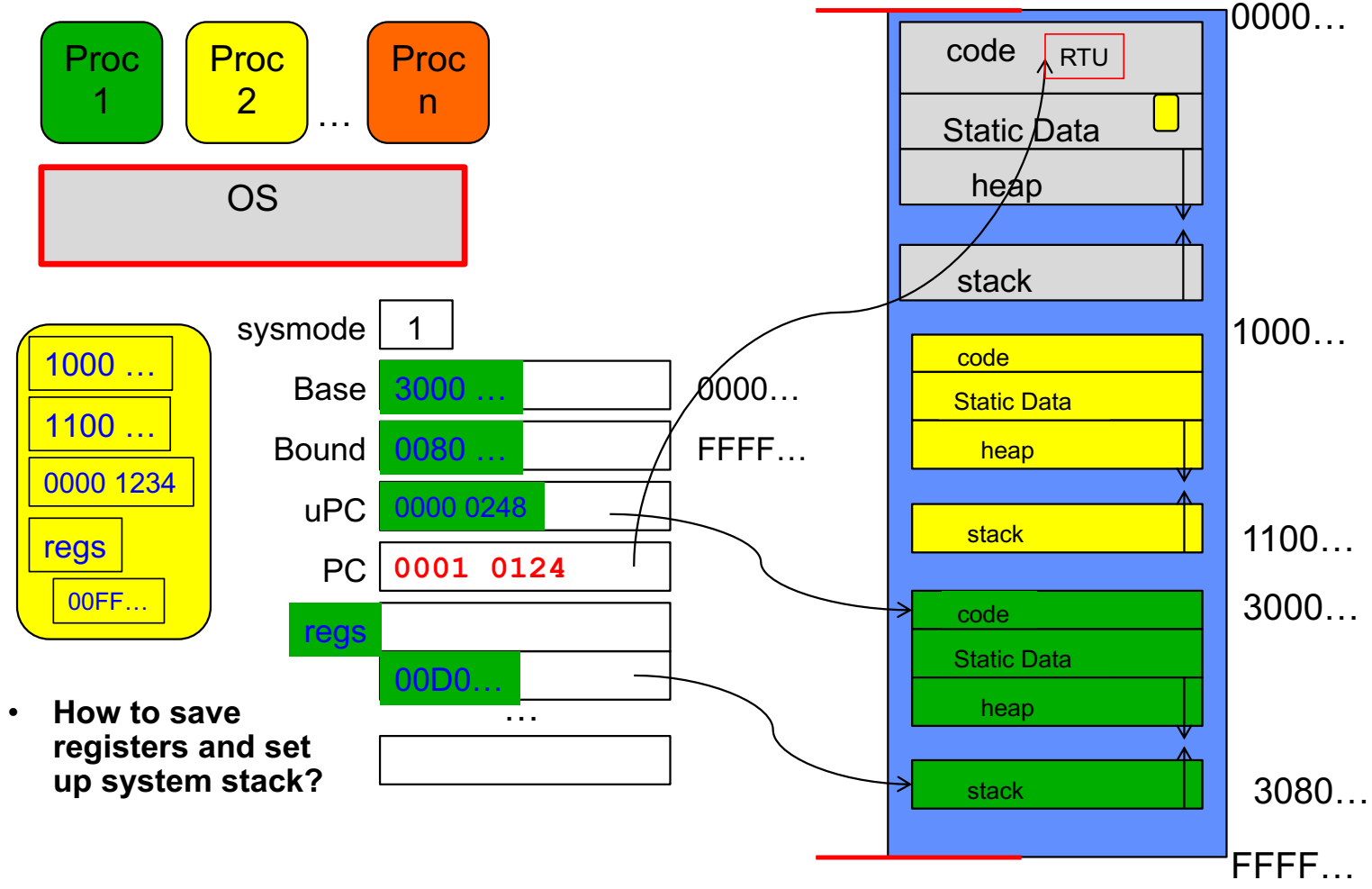
- **How to return to system?**

Simple B&B: Interrupt



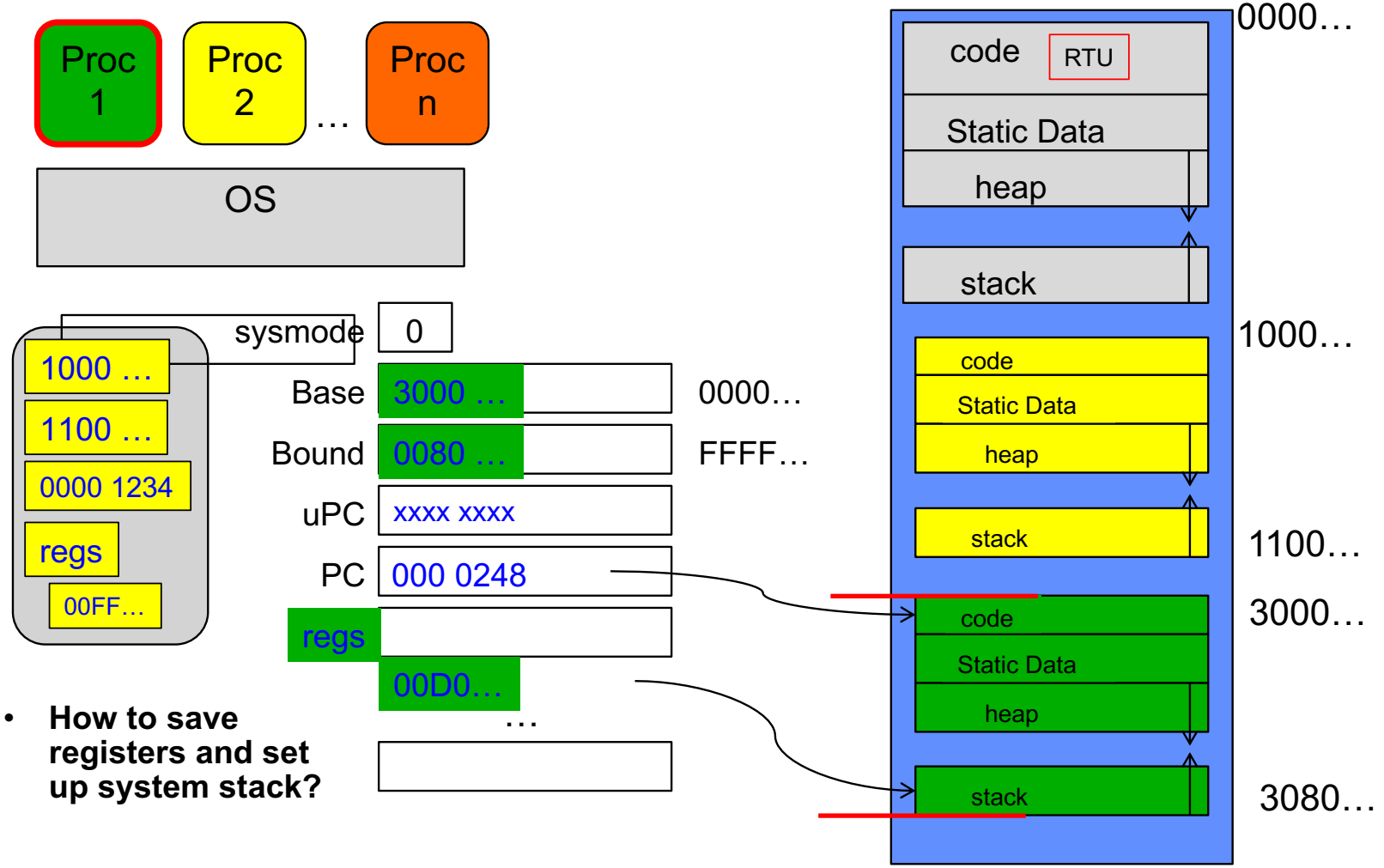
- How to save registers and set up system stack?

Simple B&B: Switch User Process



- **How to save registers and set up system stack?**

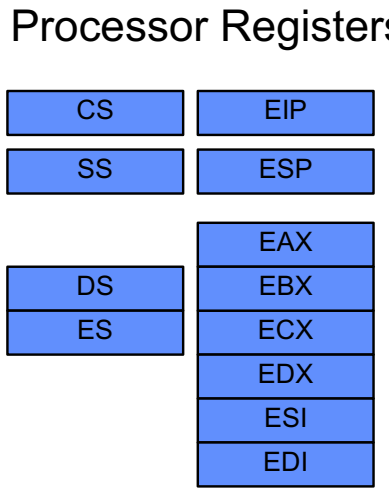
Simple B&B: “resume”



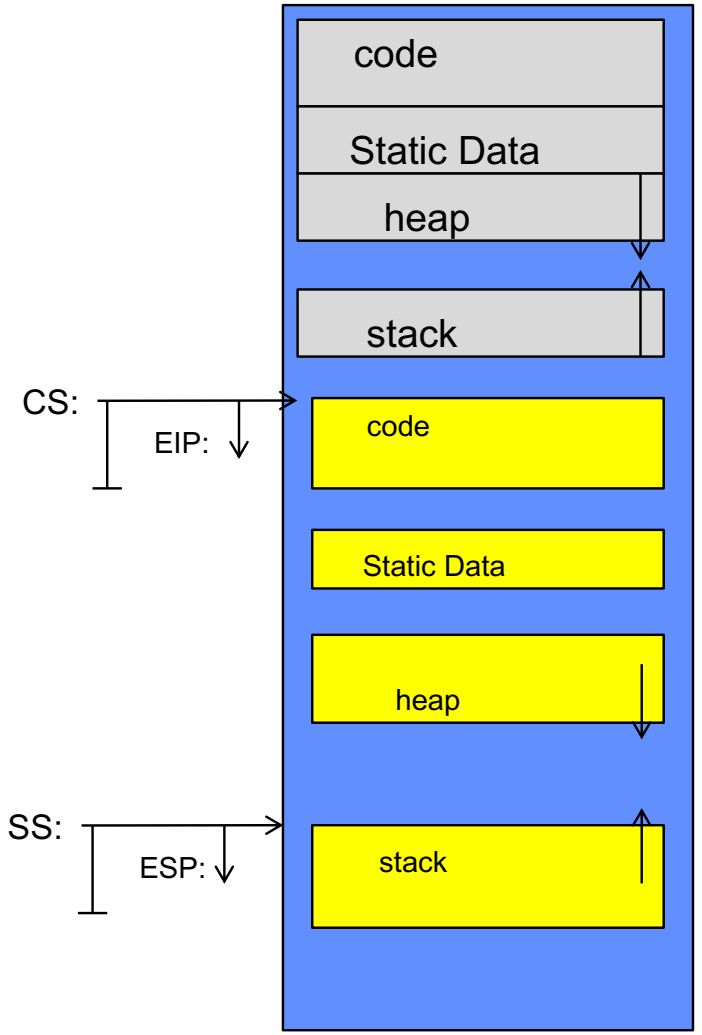
- **How to save registers and set up system stack?**

break question:
What's wrong with this simplistic address translation mechanism?

x86 – segments and stacks



Start address, length and access rights associated with each segment



Virtual Address Translation

- Simpler, more useful schemes too!
- Give every process the illusion of its own BIG FLAT ADDRESS SPACE
  - Break it into pages
  - More on this later

CPU Scheduling

Running Many Programs ???

- We have the basic mechanism to
  - switch between user processes and the kernel,
  - the kernel can switch among user processes,
  - Protect OS from user processes and processes from each other
- Questions ???
  - How do we decide which user process to run?
  - How do we represent user processes in the OS?
  - How do we pack up the process and set it aside?
  - How do we get a stack and heap for the kernel?
  - Aren't we wasting are lot of memory?
- …

Scheduling

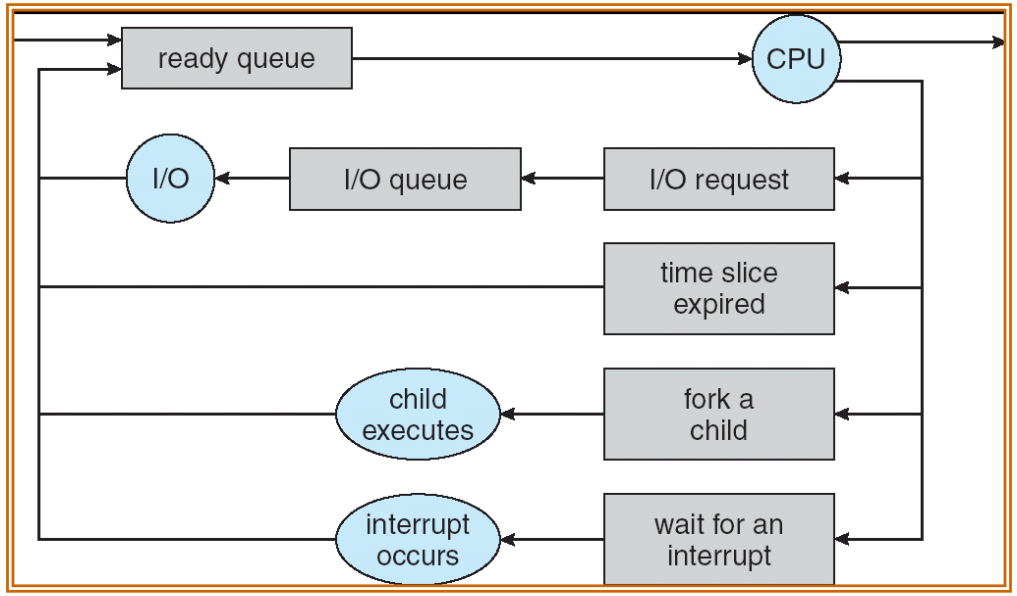
- the art, theory, and practice of deciding what to do next
- Ex: FIFO non-preemptive scheduling
- Ex: Round
- Ex: Priority
- Ex: Coordi

Definition

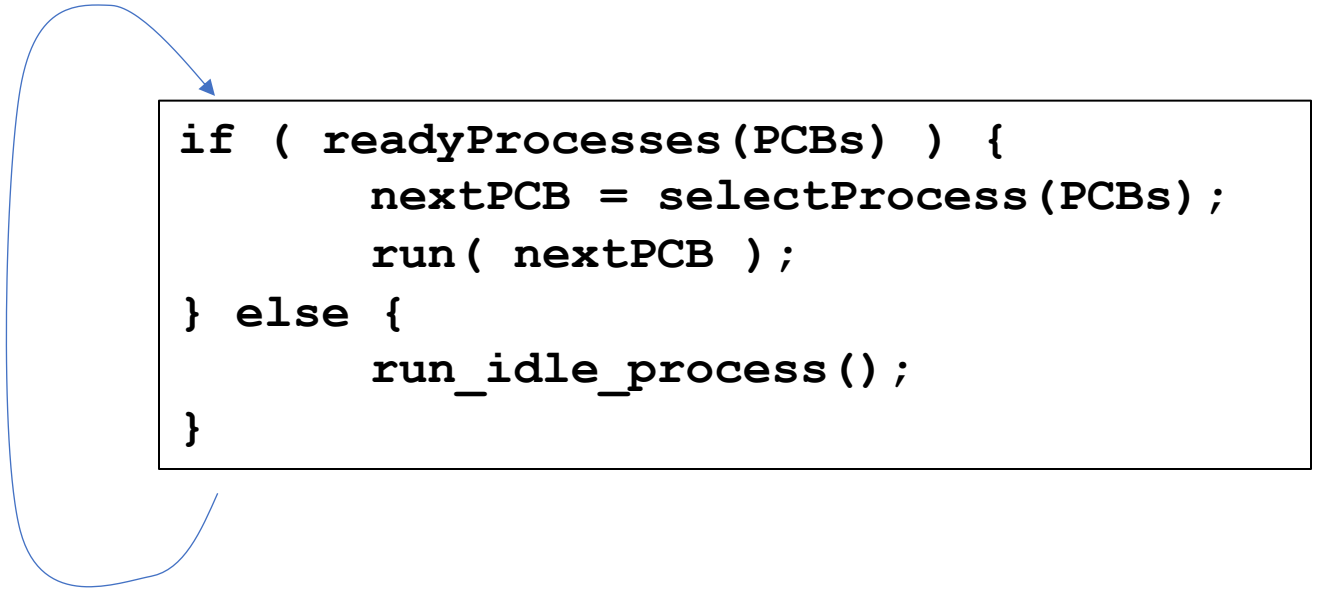
- Scheduling policy: algorithm for determining what to do next, when there are
  - multiple threads to run, or
  - multiple packets to send, or web requests to serve, or …
- Job or Task: unit of scheduling
  - quanta of a thread
  - program to completion
  - …
- Workload
  - Set of tasks for system to perform
  - Typically formed over time as scheduled tasks produce other tasks
- Metrics: properties that scheduling may seek to optimize

Processor Scheduling

- life-cycle of a thread
  - Active threads work their way from Ready queue to Running to various waiting queues.
- Scheduling: deciding which threads are given access to resources
- How to decide which of several threads to dequeue and run?
  - So far we have a single ready queue
  - Reason for wait->ready may make a big difference!



Conceptually, a Scheduler



```
if ( readyProcesses(PCBs) ) {
        nextPCB = selectProcess(PCBs);
        run( nextPCB );
} else {
        run_idle_process();
}
```

Concretely: Scheduler

- Initially a round-robin scheduler of thread quanta
- Algorithm: next_thread_to_run

```
static void schedule (void) {
  struct thread *cur = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF);
  ASSERT (cur->status != THREAD_RUNNING);
  ASSERT (is_thread (next));

  if (cur != next)
    prev = switch_threads (cur, next);
  thread_schedule_tail (prev);
}
```

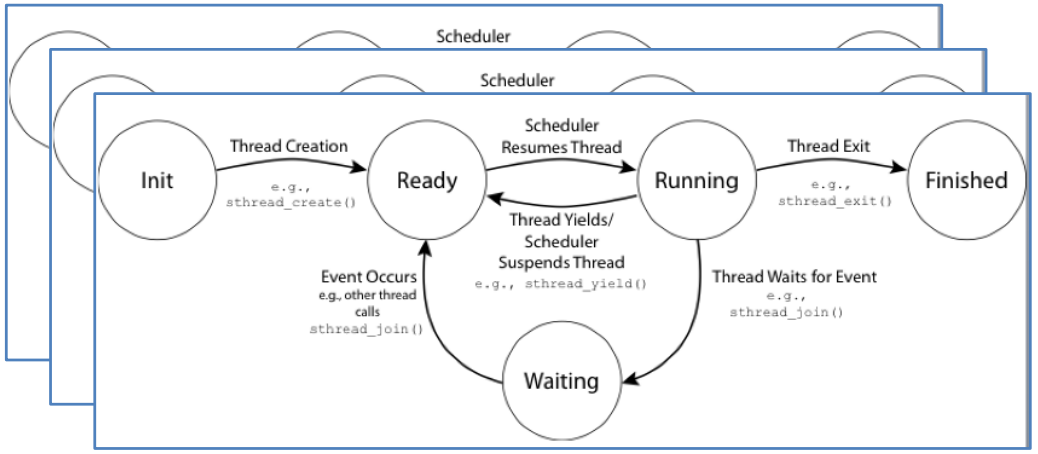
Process Control Block

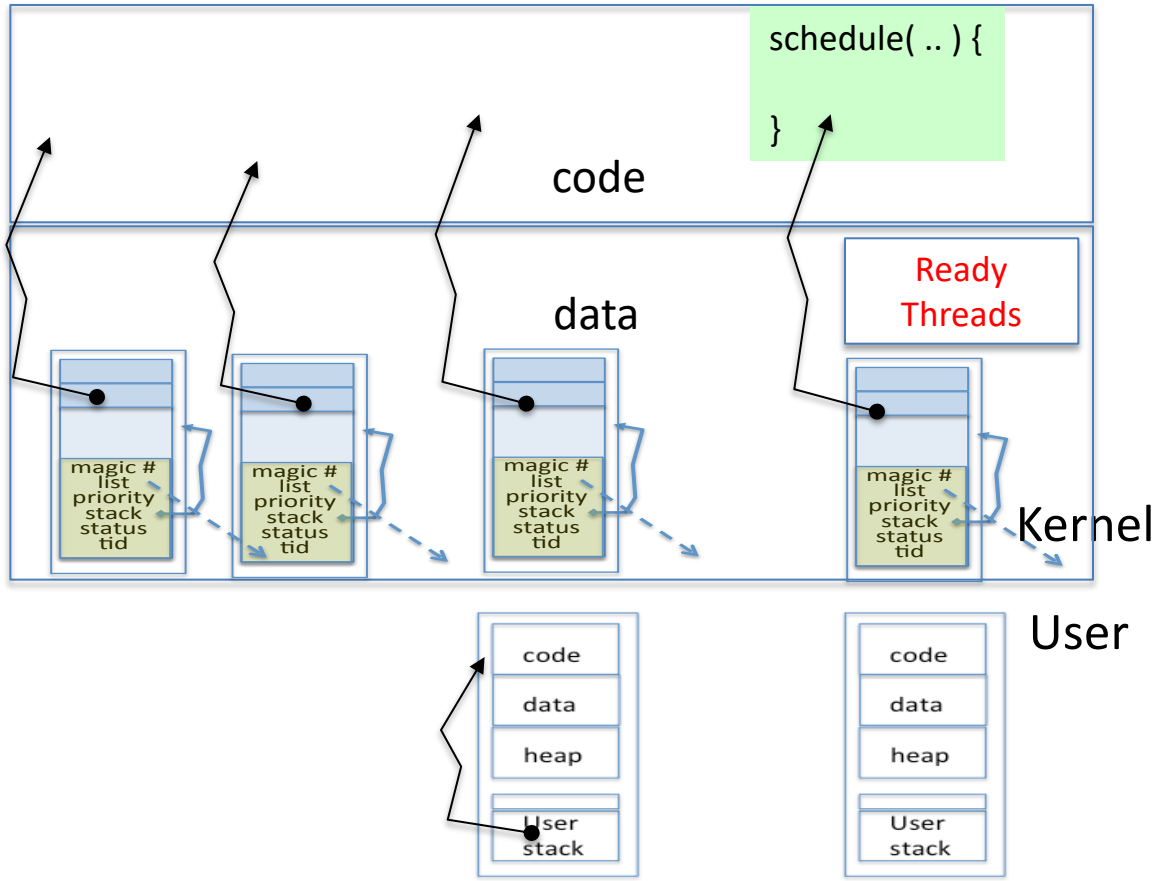
- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …
- Kernel Scheduler maintains a data structure containing the PCBs
- Scheduling algorithm selects the next one to run

Kernel threads call into scheduler

- At various points (eg. sema_down) kernel thread must block itself
  - it calls schedule to allow next task to be selected

```
void thread_block (void) {
  ASSERT (!intr_context ());
  ASSERT (intr_get_level () == INTR_OFF);
  thread_current()->status = THREAD_BLOCKED;
  schedule ();
}
```





Scheduling Metrics

- Waiting Time: time the job is waiting in the ready queue
  - Time between job's arrival in the ready queue and launching the job
- Service (Execution) Time: time the job is running
- Response Time: Time to react upon request
  - Response time is what the user sees:
    - Time to echo a keystroke in editor
  - I/O response time: time from I/O request is made to actually give response to user
- Turn-around Time (Completion Time):
  - Time between job's arrival in the ready queue and job's completion
- Throughput: number of jobs completed per unit of time
  - Throughput related to response time, but not same thing:
    - Minimizing response time will lead to more context switching than if you only maximized throughput

Scheduling Policy Goals/Criteria

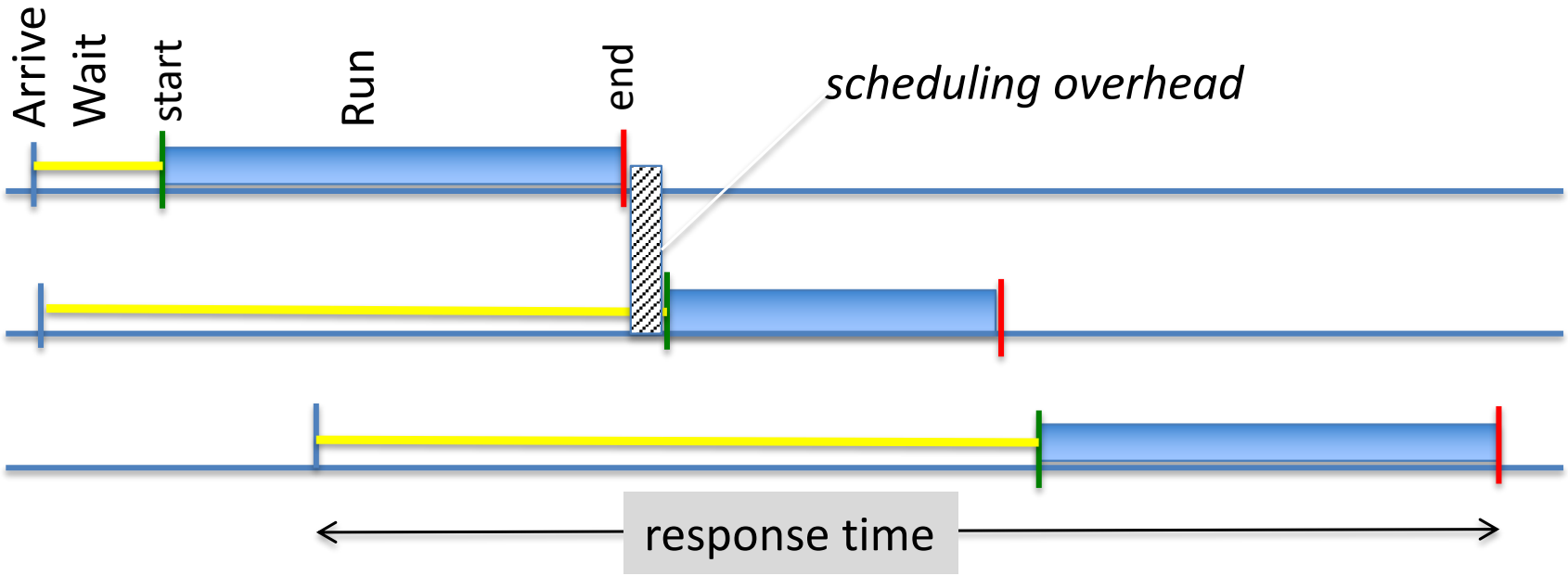
- Minimize Response Time
  - Minimize elapsed time to do an operation (or job)
- Maximize Throughput
  - Two parts to maximizing throughput
    - Minimize overhead (for example, context-switching)
    - Efficient use of resources (CPU, disk, memory, etc)
- Fairness
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    - Better *average* response time by making system *less* fair

Different systems have different goals

- Multi-user systems
  - Fairness: giving each process a fair share of CPU
  - Balance: keeping all parts of systems busy
- Batch systems
  - Throughput: maximize jobs per hour
  - Turn-around time: minimize time between submission & completion
  - CPU utilization: keep CPU busy all the time
- Interactive systems
  - Response time: respond to user requests quickly
  - Proportionality: meet users' expectations
- Real-time systems
  - Meeting deadlines: avoid losing data
  - Predictability: avoid quality degradation in multimedia systems

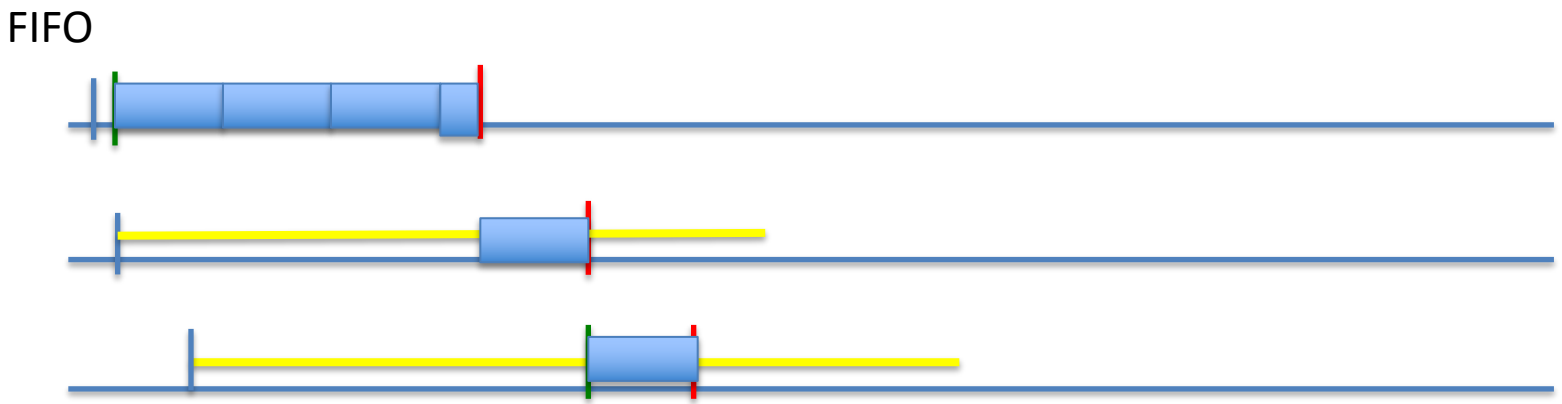
First In First Out - FCFS

- Schedule tasks in the order they arrive
  - Run until they complete or give up the processor



Why no FIFO? The convoy effect

- Longer job comes first, and
  all the other jobs wait until it is completed
  - Waiting time is accumulated
  - Once completed, the job does not need to wait
- Late-comer takes the all penalty, which might be your job

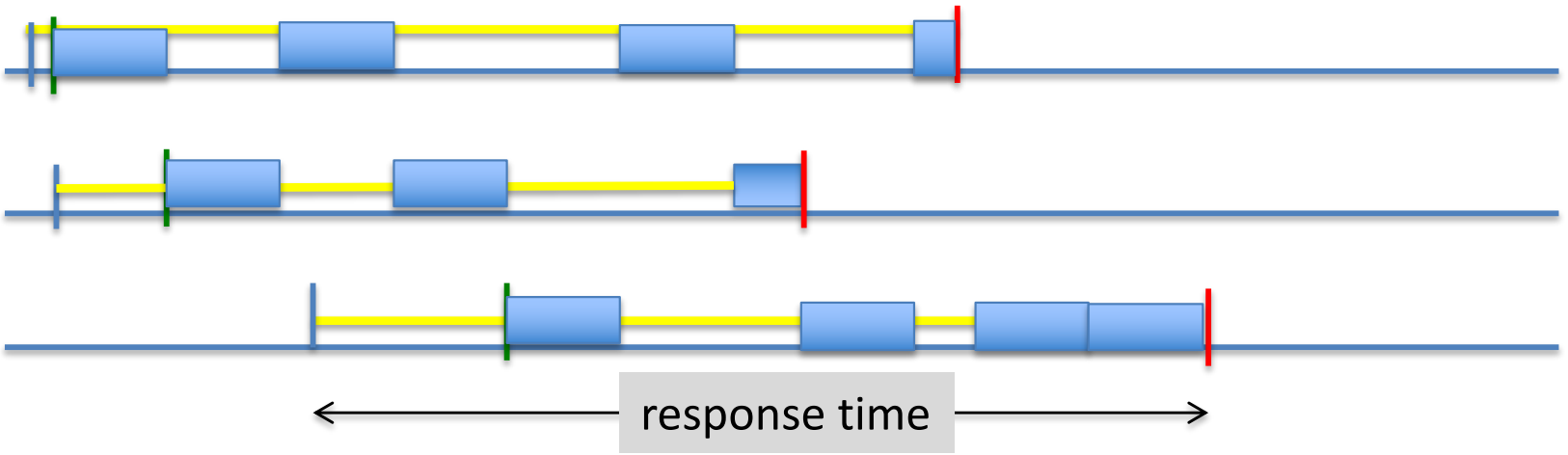


What if we Knew the Future?

- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
- 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
    - but how do you now???
- 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
- Want a simple approximation to SRTF …

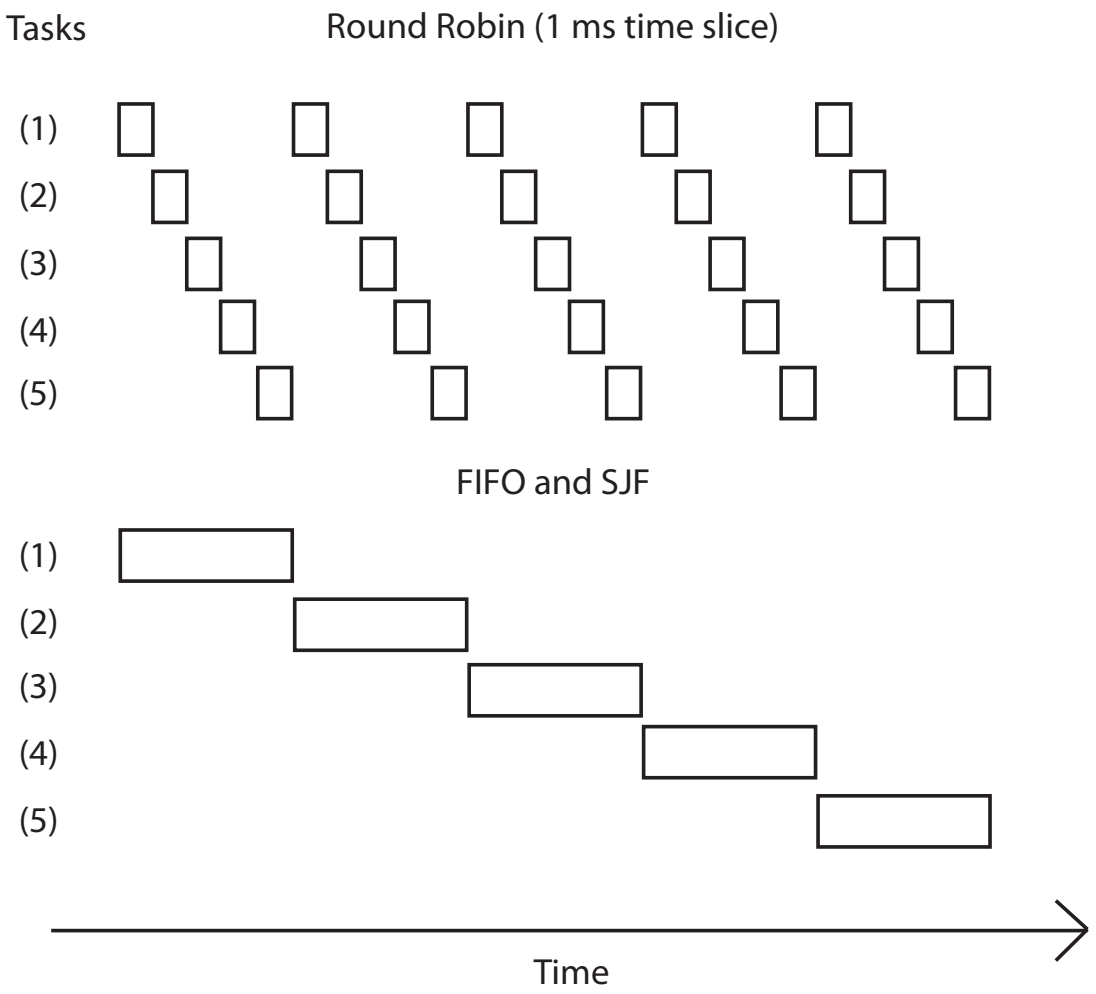
Round-Robin

- Each task gets a fixed amount of the resource (time quantum)
  - if it does not complete, goes back into queue



- How large a time quantum?
  - Too short? Too long? Trade-offs?

Round Robin vs. FIFO



How RR FIFO are different?

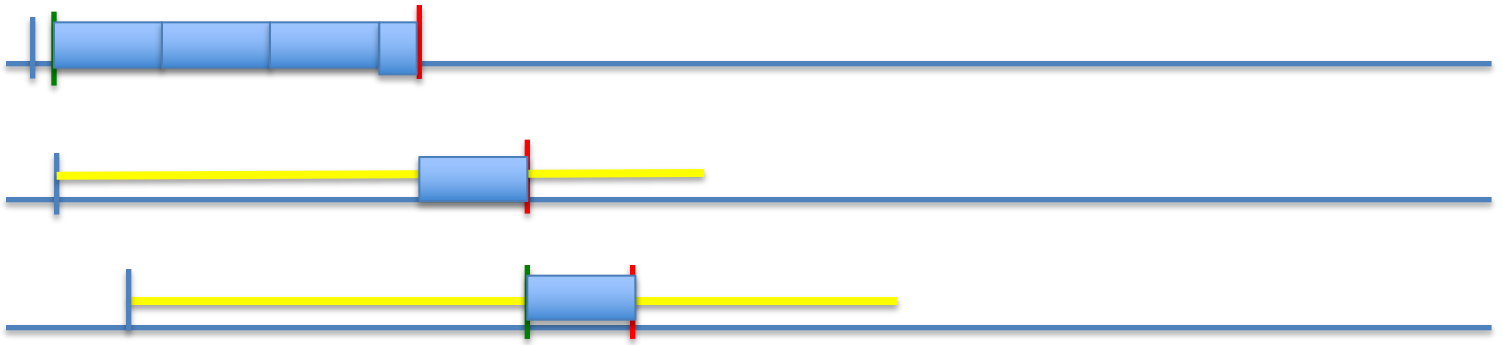
- Utilization
- Turn-around time or waiting time
- Then, what is RR for?
  - Fairness in short time window?
  - What if longest job comes first?
  - How long do you have to wait until you get CPU share?

An example:

- P (arrival, exec.time)
  - in ticks
- Regarding the following three processes,
  - P1 (2, 10)
  - P2 (1, 5)
  - P3 (0, 7)
- Present the execution when we use FIFO, SJF, SRTF, RR (2)
  - RR with time quantum 2 ticks
- Waiting time when we use FIFO? SJF? SRTF? RR?

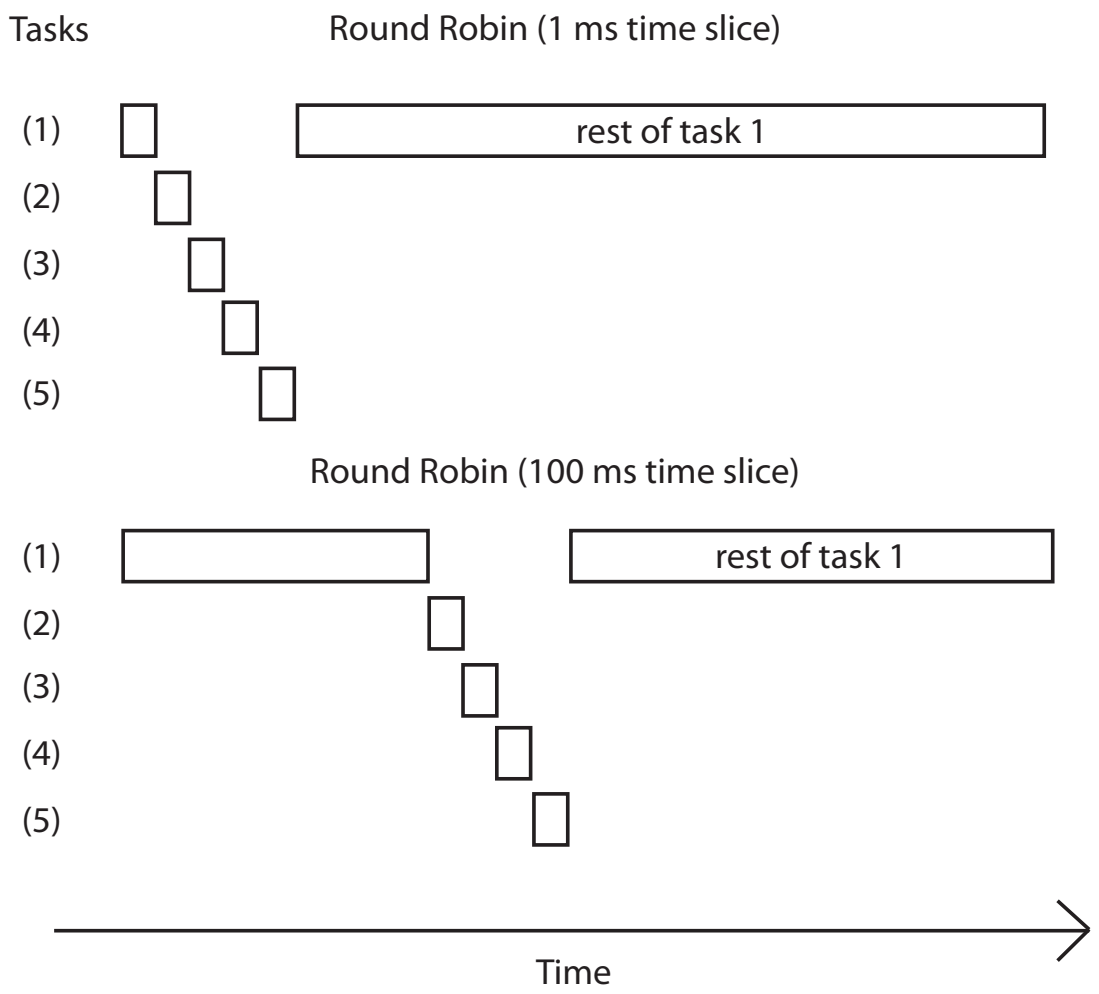
Round Robin vs FIFO

FIFO



Round Robin

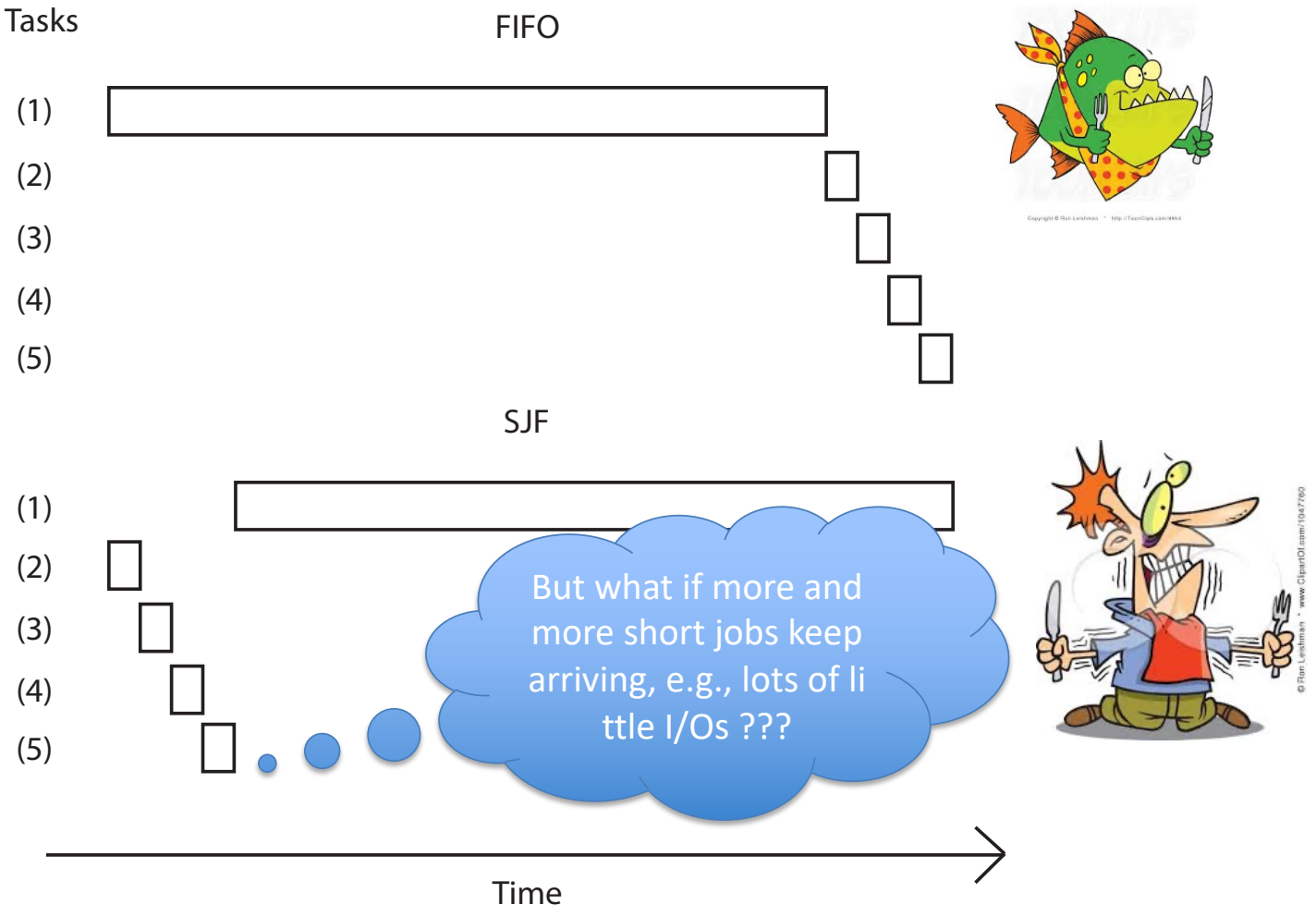
Round Robin Slice



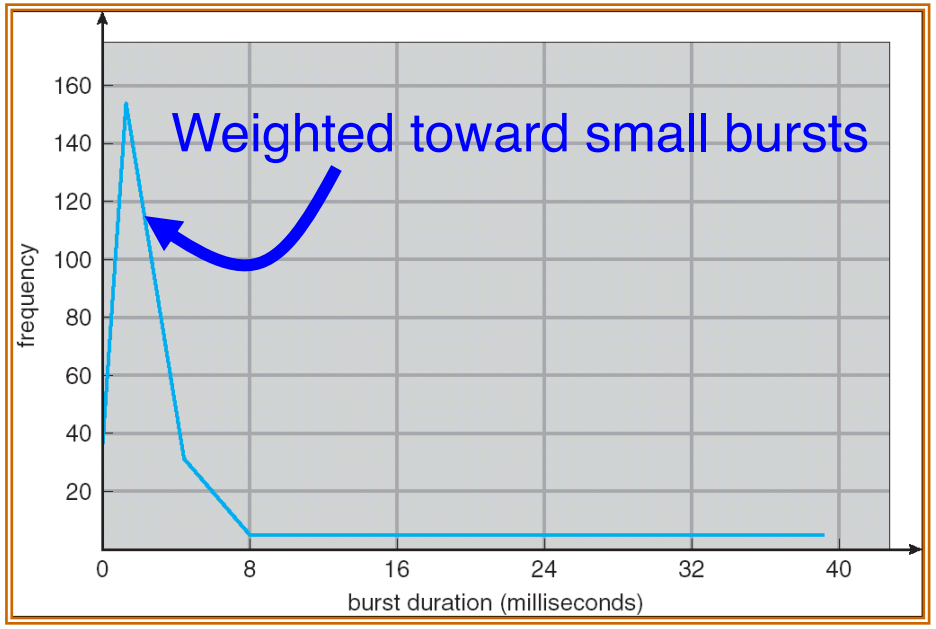
Round-Robin Discussion

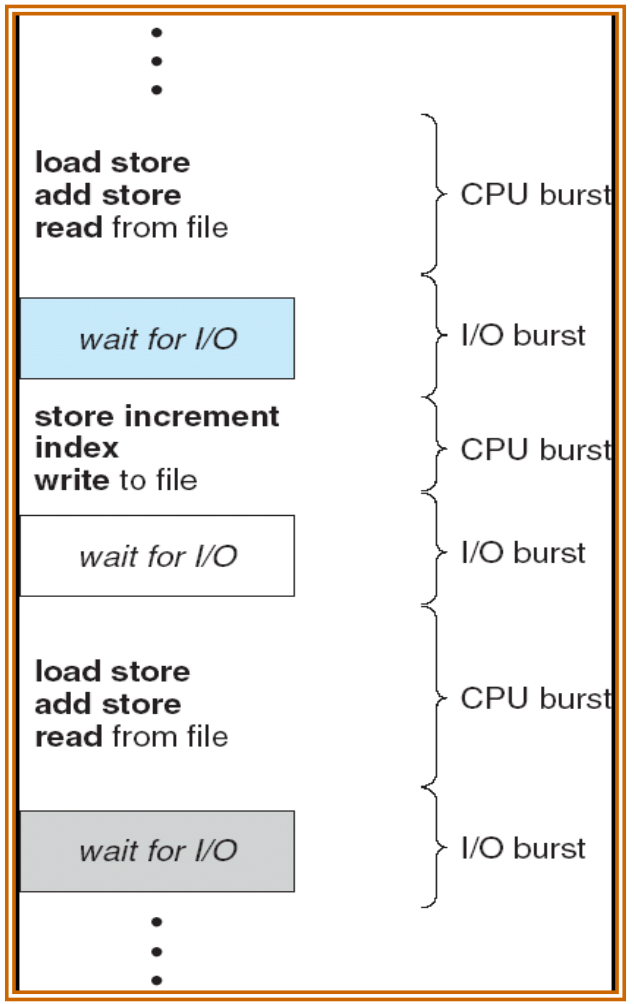
- How do you choose time slice?
  - What if too big?
    - Response time suffers
  - What if infinite (∞)?
    - Get back FCFS/FIFO
  - What if time slice too small?
    - Throughput suffers!
- Actual choices of timeslice:
  - Initially, UNIX timeslice one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo each keystroke!
  - In practice, 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

FIFO vs. SJF


Tasks
FIFO
(1)
(2)
(3)
(4)
(5)
SJF
(1)
(2)
(3)
(4)
(5)
But what if more and more short jobs keep arriving, e.g., lots of li ttle I/Os ???
Time

CPU Bursts





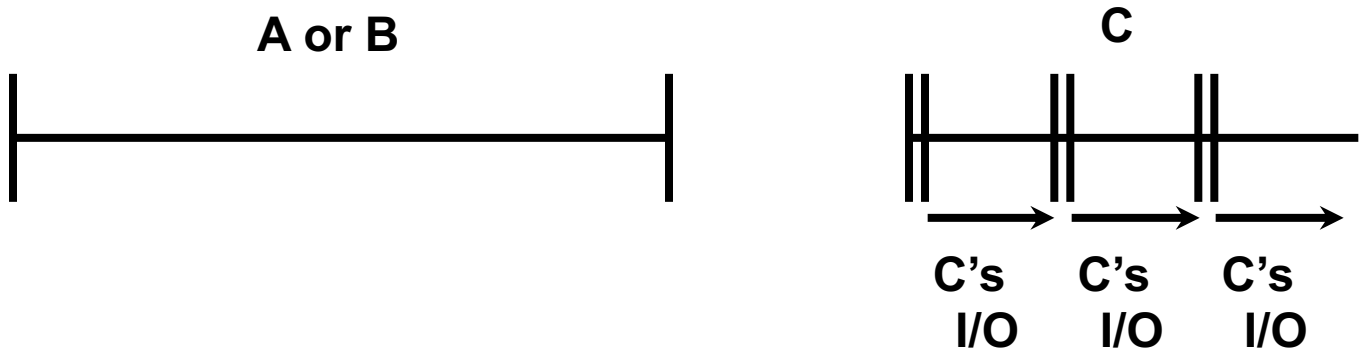
- Programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With timeslicing, a thread may be forced to give up CPU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

Discussion

- SJF/SRTF are best 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

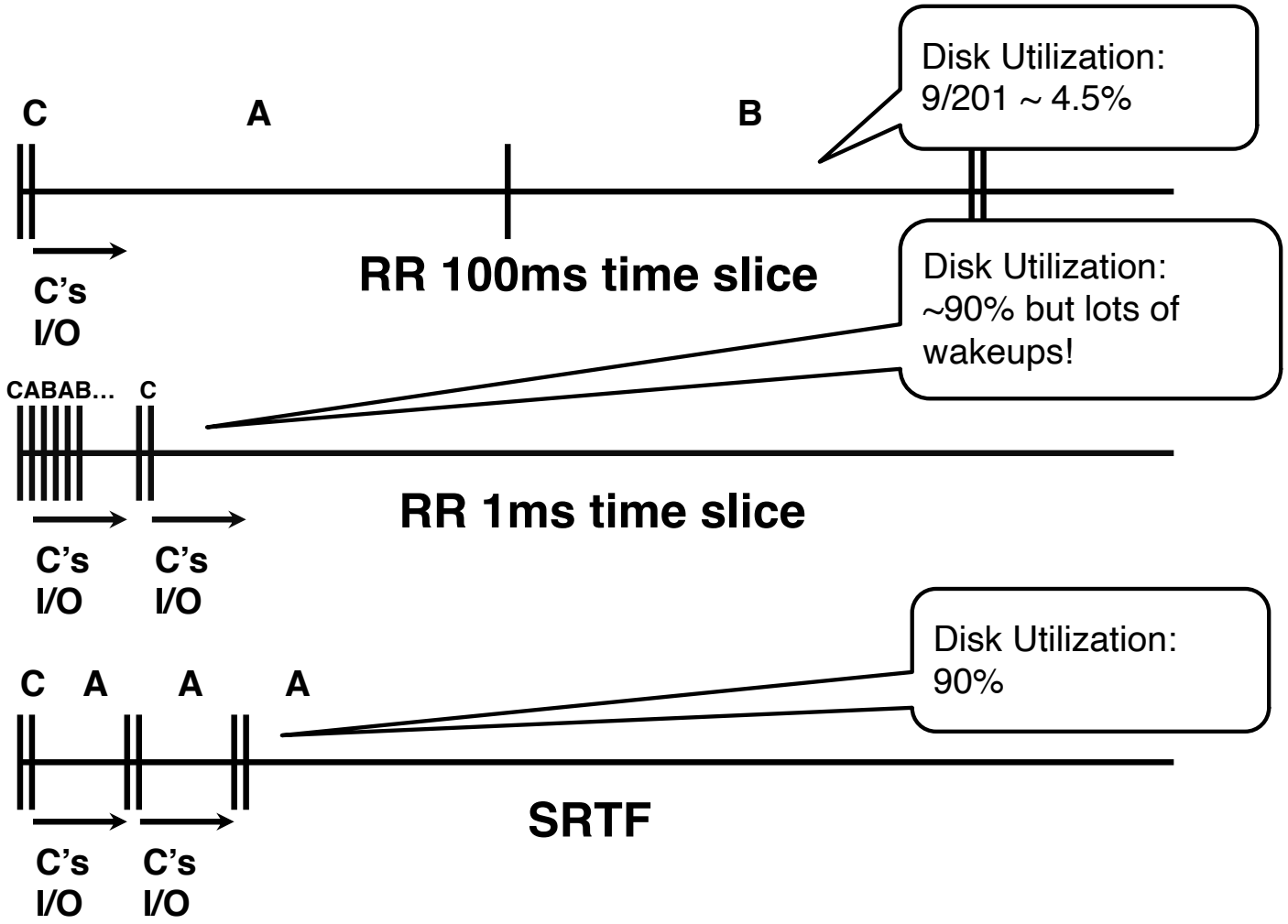
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - SJF becomes the same as FCFS (i.e., FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - SRTF (and RR): short jobs not stuck behind long ones

Example to illustrate benefits of SRTF



- Three jobs:
  - A,B: CPU bound, each task runs 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


C
A
B
Disk Utilization:
9/201 ~ 4.5%
C's
I/O
RR 100ms time slice
Disk Utilization:
~90% but lots of
wakeups!
CABAB...
C
C's
I/O
C's
I/O
RR 1ms time slice
C
A
A
A
Disk Utilization:
90%
C's
I/O
C's
I/O
SRTF

SRTF Further discussion

- Starvation
  - SRTF can lead to starvation if there are 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: even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
  - However, can use SRTF as a yardstick for measuring other policies
  - Optimal => Practical approximations?
- SRTF Pros & Cons
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Unfair (-)

Priority Scheduling

- Priorities can be a way to express desired outcome to the scheduler
  - important (high priority) tasks first, quicker, …
  - while low priority ones when resources available, …

- How might priorities interact positively / negatively with synchronization? With I/O ?

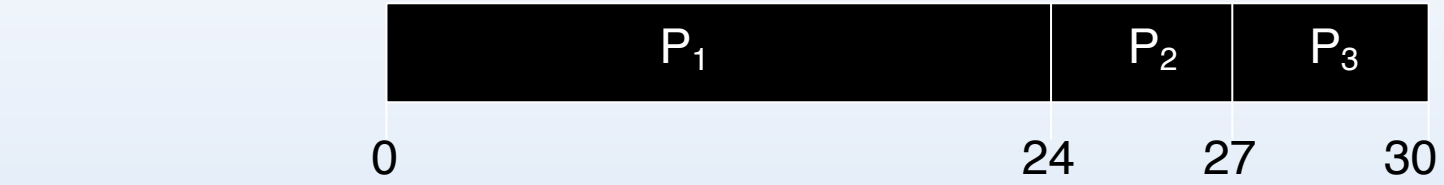
First-Come, First-Served (FCFS) Scheduling

- 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:

Process	Burst Time
P_1	24
P_2	3
P_3	3

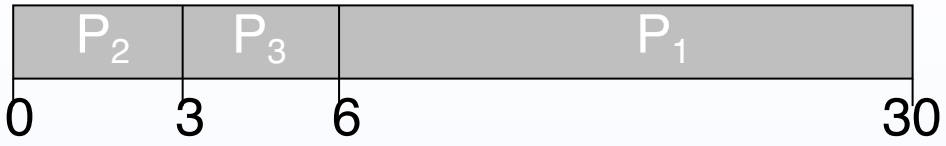
  - Suppose processes arrive in the order: P_1 , P_2 , P_3 The Gantt Chart for the schedule is:



  - 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 behind long process

FCFS Scheduling (Cont.)

- Example continued:
  - Suppose that processes arrive in order: P_2 , P_3 , P_1
    Now, the Gantt chart for the schedule is:



  - Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
  - 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)
- FCFS Pros and Cons:
  - Simple (+)
  - Short jobs get stuck behind long ones (-)
    - Safeway: Getting milk, always stuck behind cart full of small items

Round Robin (RR)

- 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
  - 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
- Performance
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ Interleaved
  - 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	Remaining Time
P_1	53	53
P_2	8	8
P_3	68	68
P_4	24	24

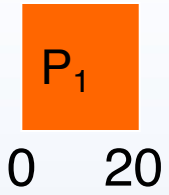
  - The Gantt chart is:

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	33
P_2	8	8
P_3	68	68
P_4	24	24

  - The Gantt chart is:



Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	33
P_2	8	0
P_3	68	68
P_4	24	24

  - The Gantt chart is:

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time	
P_1	53	33	
P_2	8		0
P_3	68	48	
P_4	24	24	

  - The Gantt chart is:

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	33
P_2	8	0
P_3	68	48
P_4	24	4

  - The Gantt chart is:

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	13
P_2	8	0
P_3	68	48
P_4	24	4

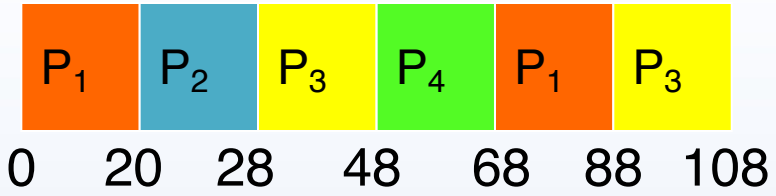
  - The Gantt chart is:

Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	13
P_2	8	0
P_3	68	28
P_4	24	4

  - The Gantt chart is:

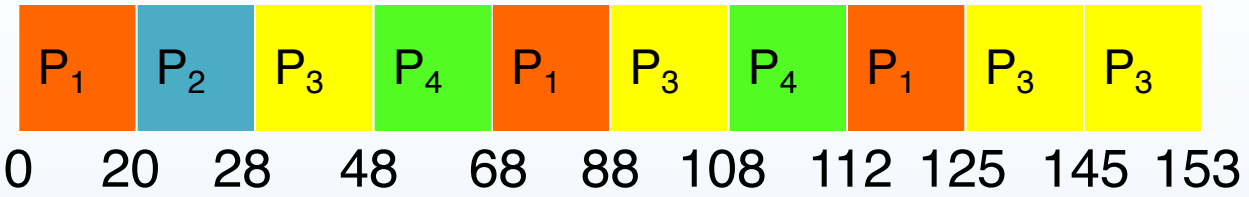


Example of RR with Time Quantum = 20

- Example:

Process	Burst Time	Remaining Time
P_1	53	0
P_2	8	0
P_3	68	0
P_4	24	0

  - The Gantt chart is:



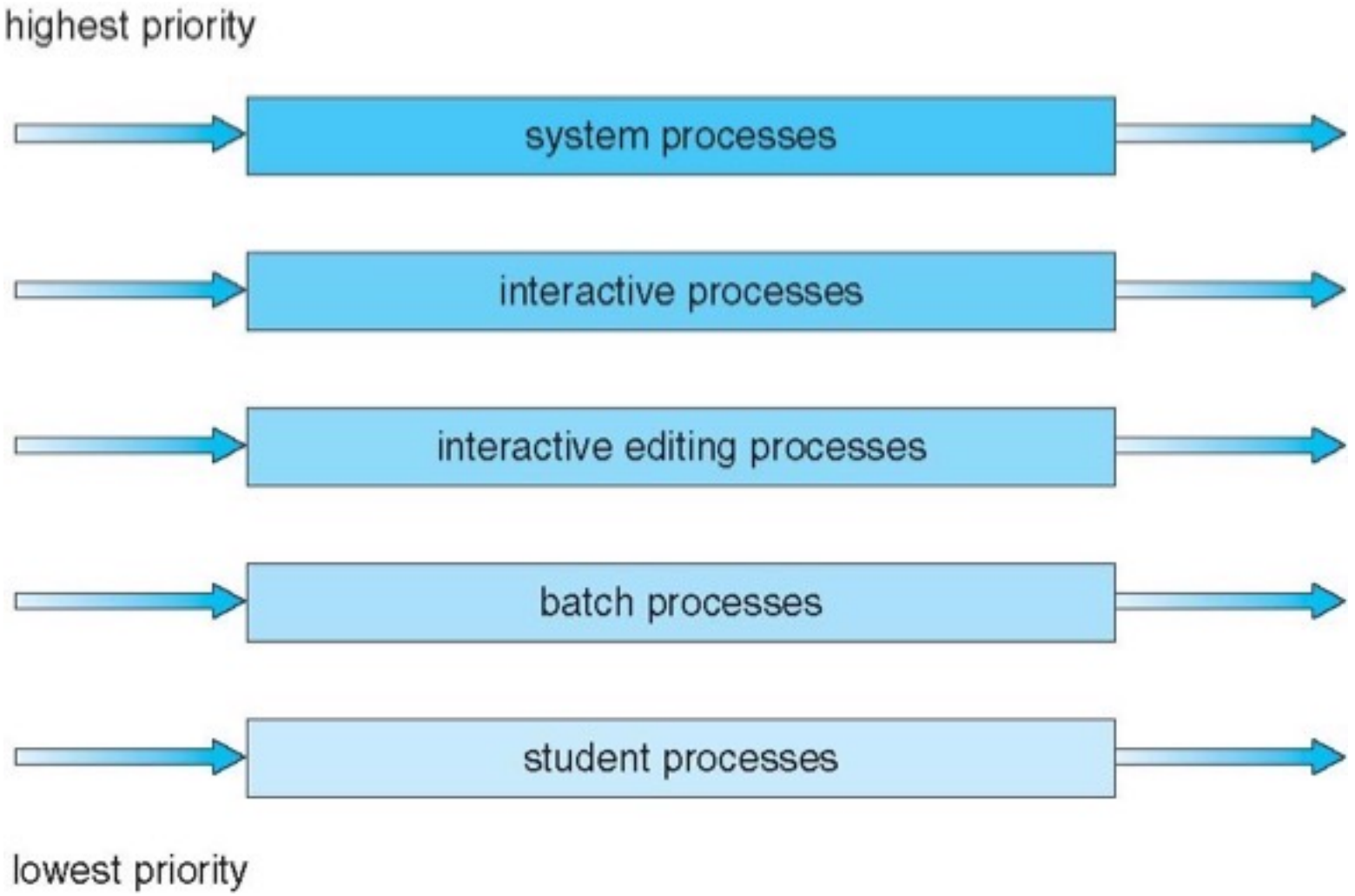
  - Waiting time for P_1=(68-20)+(112-88)=72
    P_2=(20-0)=20
    P_3=(28-0)+(88-48)+(125-108)=85
    P_4=(48-0)+(108-68)=88
  - Average waiting time = (72+20+85+88)/4=66¼
  - Average completion time = (125+28+153+112)/4 = 104½

- Thus, Round-Robin Pros and Cons:
  - Better for short jobs, Fair (+)
  - Context-switching time adds up for long jobs (-)

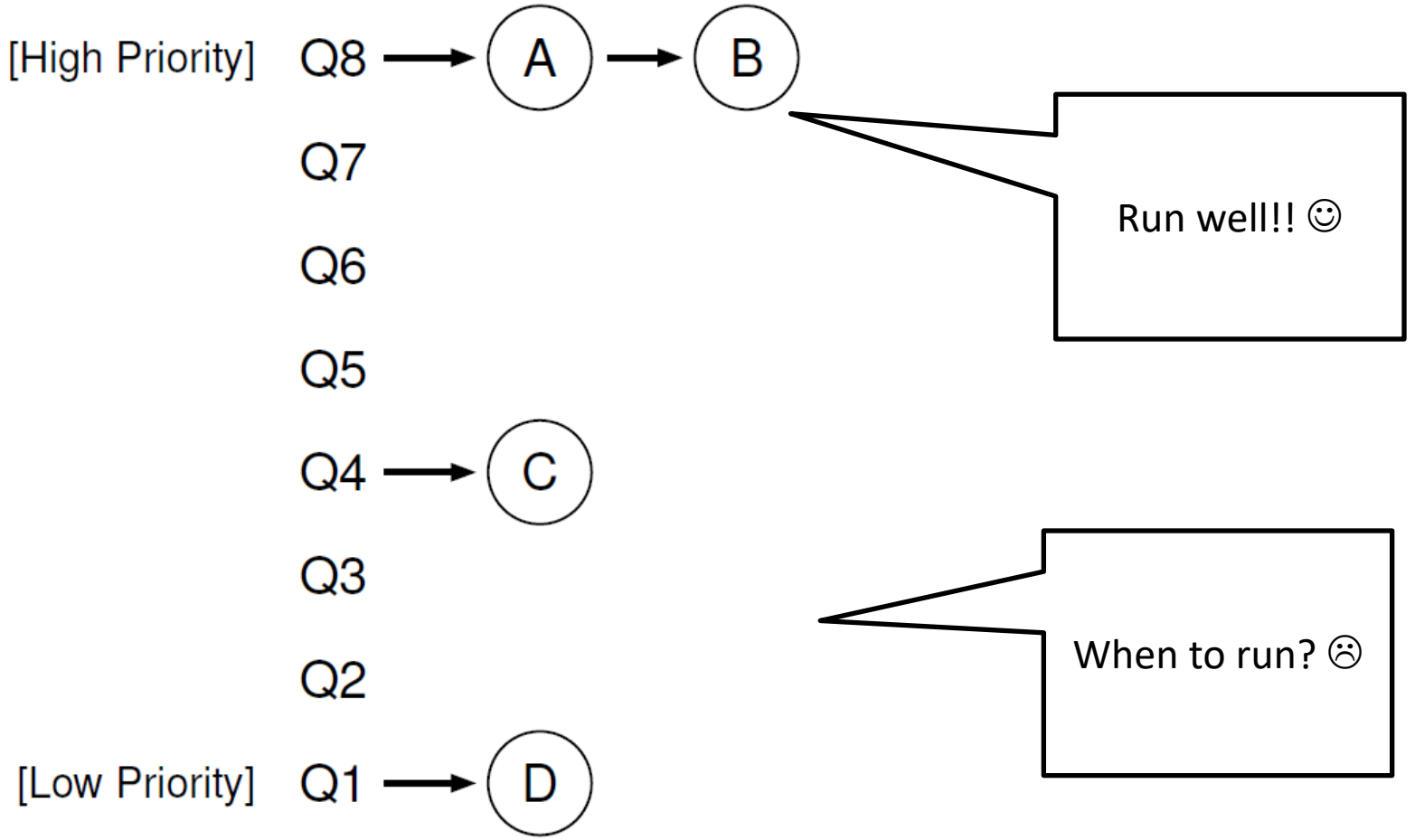
Multi-Level Queue (MLQ)

- Multiple ready queues for different purposes
  - **E.g. foreground** (interactive) with RR, **background** (batch) with FCFS
- Process remaining in the queue
- Each queue has its own scheduling policy
- Scheduling among queues required
  - Fixed priority scheduling
    - Highest priority queue always takes CPU
  - Time slice given to a queue
    - Each queue has its own time quantum
    - E.g. 80% to foreground in RR, 20% to background in FCFS

Example of multi-level queue



A problem in MLQ



Multi-Level Feedback Queue (MLFQ)

- MLFQ used in ancient CTSS (compatible time-sharing system), Multics OS
- Fundamental ideas are used in modern OS (Linux, Windows, Solaris)
- Goals of MLFQ
  - Optimize for turn-around time (TAT)
    - SJF/STCF are impractical
  - Optimize for response time
    - RR is bad for turn-around time
    - Job length has to be estimated in order to optimize turn-around time

Rules for MLFQ

- Process selection algorithm
  - Priority level is assigned to each queue
  - Next process is determined according to the following rules

    - **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't).
    - **Rule 2:** If Priority(A) = Priority(B), A & B run in RR.

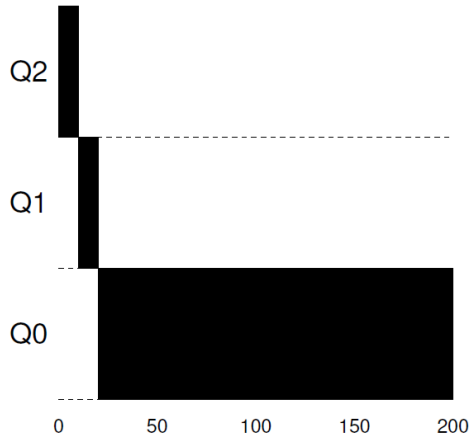
- Priority can be changed along with the execution, scheduler has to monitor the process
  - E.g. Interactive process,
    - Wait until getting keyboard input, low CPU utilization → keeps priority high
  - E.g. CPU intensive process,
    - Long-running CPU consuming process, → lowers down priority
- How?
  - Observed behaviors → Make a history → learn and predict the future behavior

MLFQ: How to change priority?

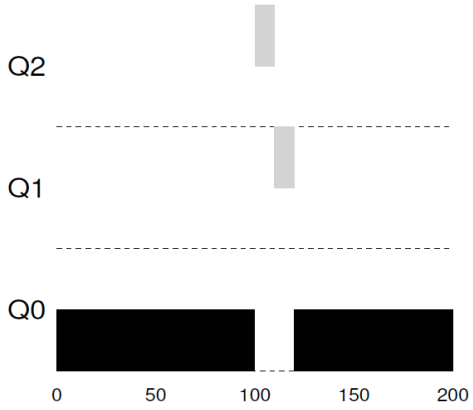
- Workload assumptions
  - Mixed workload
  - Short-running job: small CPU execution time → interactive process
    - response time is important
  - Long-running job: longer CPU execution time →CPU-bound jobs
- Priority changing rules
  - **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
  - **Rule 4a:** If a job uses up an entire time slice while running, its priority is *reduced* (i.e., it moves down one queue).
  - **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the *same* priority level.

MLFQ: Examples

- E.g. #1 A single long running process (A)
  - Time slice: 10ms
  - After creation, place on Q2 → Rule 3
  - Time slice expires, lower down on Q1 → Rule 4a
  - Rule 4a applied again, place down to Q0
- E.g. #2 Along came a short job (B)
  - Time slice: 10ms
  - While running A, B enters the system, B is short-running interactive process (B, 20 ms)
  - Place B on Q2 → Rule 3
  - B consumes all time slice, lower down to Q1 → Rule 4a
  - After the completion of B, A runs in Q0
- MLFQ: does not know whether a process is long-running or short-running, but
  - Short job → quit after quick execution
  - Long job → smoothly down the queues → Batch like



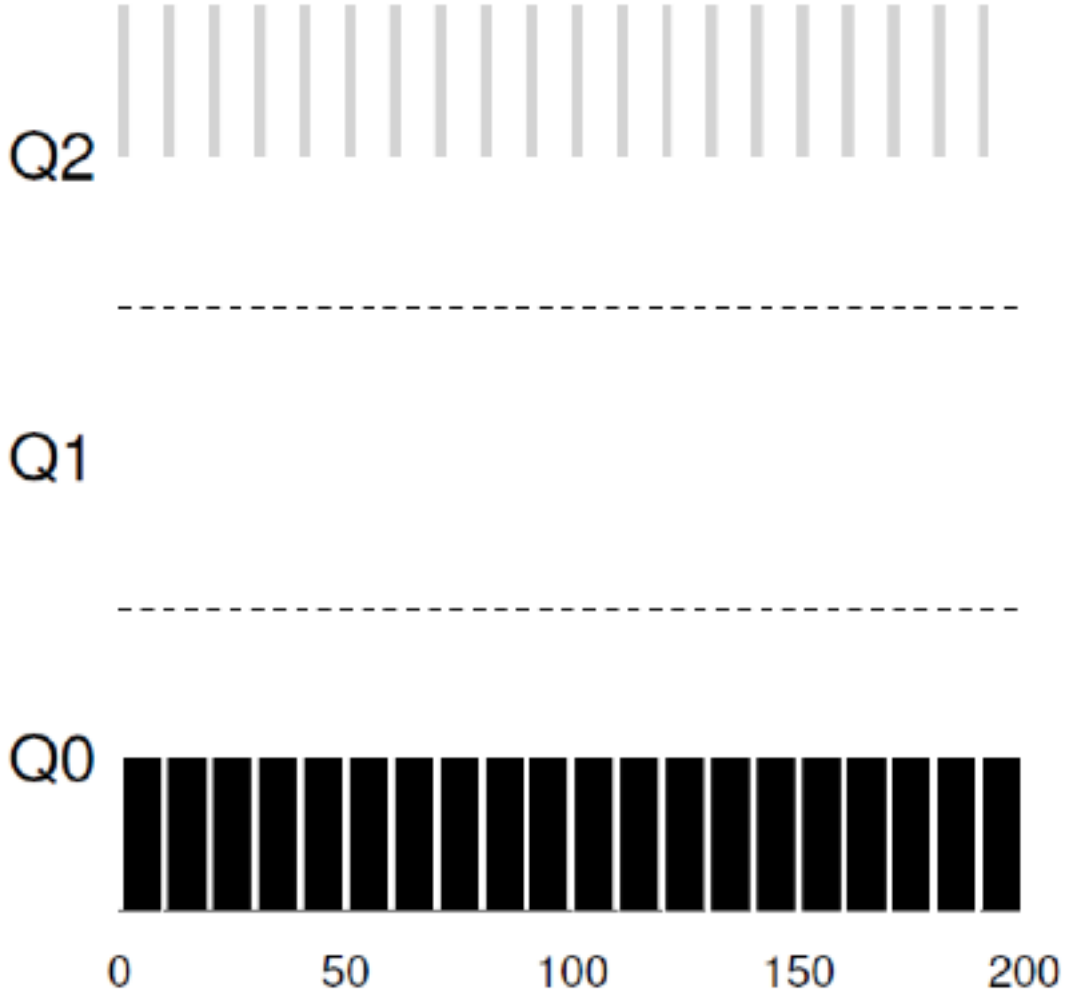
A long running process



A short job

MLFQ: I/O mixed

- E.g. #3 I/O mixed (A)
  - Time slice: 10ms
  - After creation, place A on Q2 → Rule 3
  - Before the time slice expiration, begin I/O waits in queue, yield CPU (used only 1ms), Keep in the priority Q2 → Rule 4b
  - Lower priority job runs until I/O completes, (Q0) → CPU - I/O overlap
  - After time slice expiration, execute higher priority job (Q2) → Rule 4b
- Interactive process: fast response guarantee

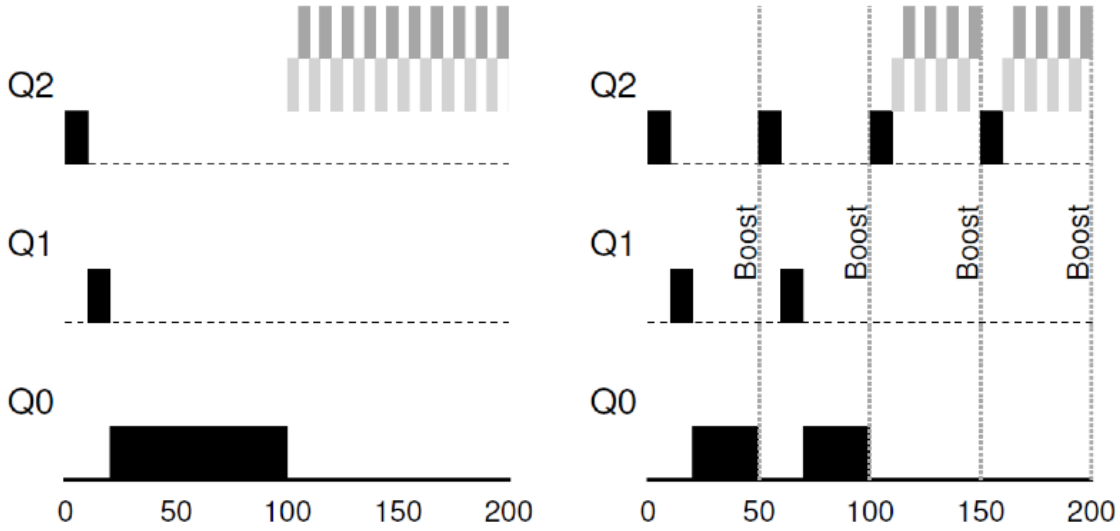


MLFQ is not perfect

- MLFQ good for
  long-running CPU-bound process and
  short running I/O interactive process
- Yet, starvation
  - Too many interactive processes → Can long-running processes receive CPU?
- CPU can be monopolized
  - Malicious user can break into the fair scheduling policy
    - E.g. Use CPU enough (99%) and then request I/O just before the time slice expiration
- A real-world program cannot be clearly separated into CPU-bound, I/O interactive jobs
  - Even in a single program, different phase exist

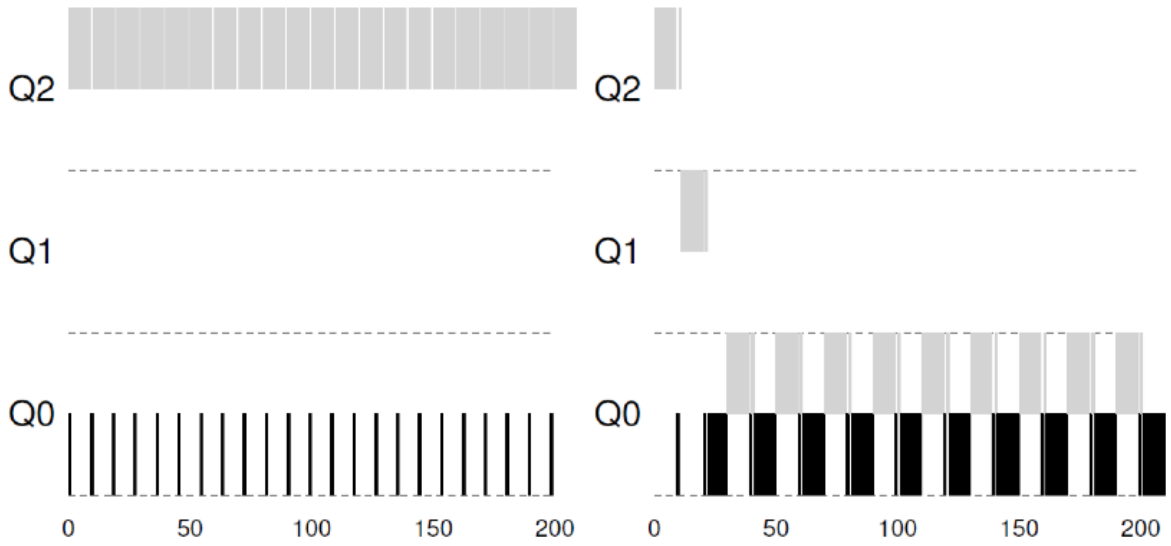
Improved MLFQ

- Priority boost
  - Periodically increase the priority of all jobs → resolves starvation
    - **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.
  - At the highest queue, schedule through RR
  - CPU-bound process changes into interactive ones?
    - Handle the job after boost
- E.g. #4
  - No boost, Boost every 50ms
- S
  - Too high → Long running process can starve
  - Too low → interactive process can get proper share of CPU



Another improvement to MLFQ

- Accounting of CPU time
  - Actual CPU time is accounted accurately
  - Instead of time slice
- Rule 4a + 4b → Rule 4
  - **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
  - Regardless of the yields of CPU, a process is de-prioritized when it uses up CPU time (time allotment)
- E.g. #5
  - Rule 4a, 4b (left), Rule 4 (right)

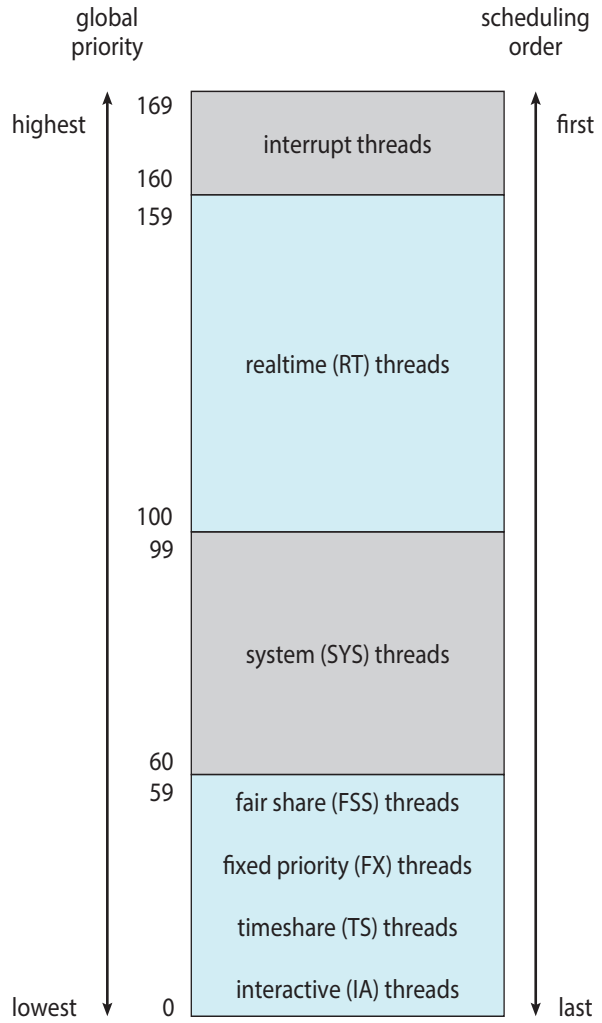


MLFQ case: Solaris OS

- Priority-based preemptive scheduling
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

Solaris MLFQ

- Solaris dispatch table
  - Time slice (quantum)
  - queue down condition
  - boost condition is given
- Priority boosted every 1s



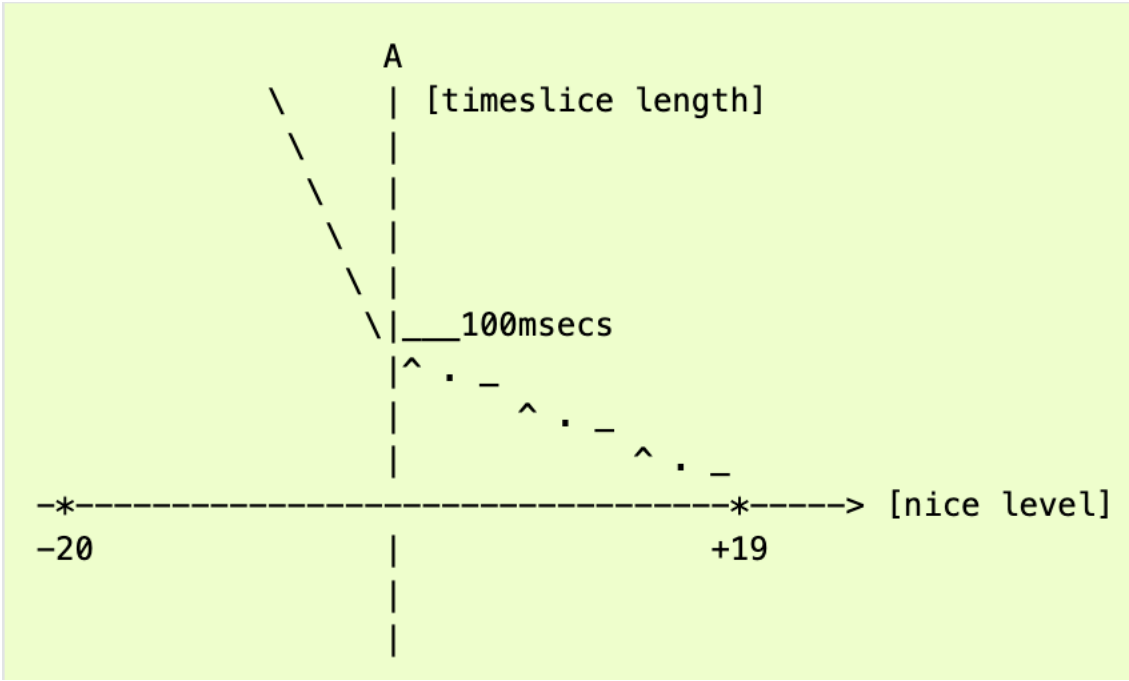
Lowest Priority

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Highest Priority

Unix/Linux Scheduling: old-fashioned

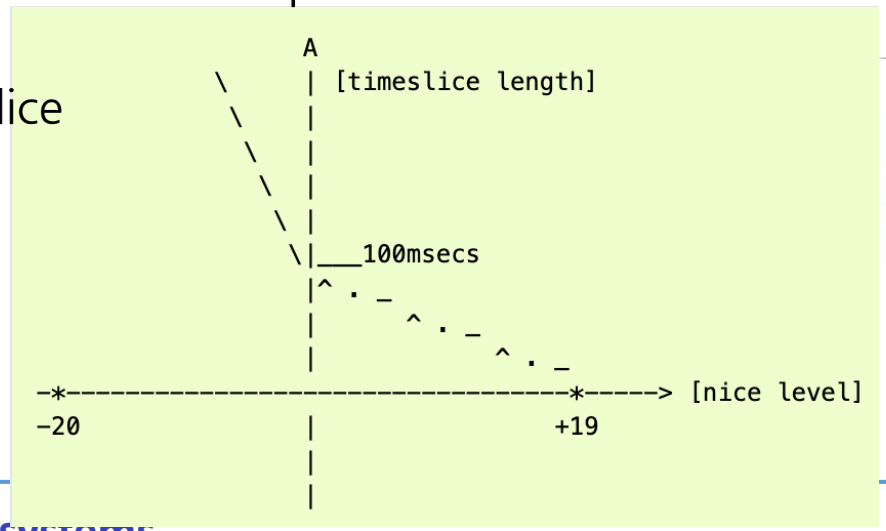
- round-robin with nice value
- nice: how much you're nice to other processes (less utilization, less quantum)
- nice value: -20 ~ +19
  - 19 gets 10 ms quantum
  - -20 gets 400 ms quantum
  - static granularity called tick or jiffy
- O(1) scheduler (later slide)
  - +19 gets 5ms (for 1000 HZ ticks)

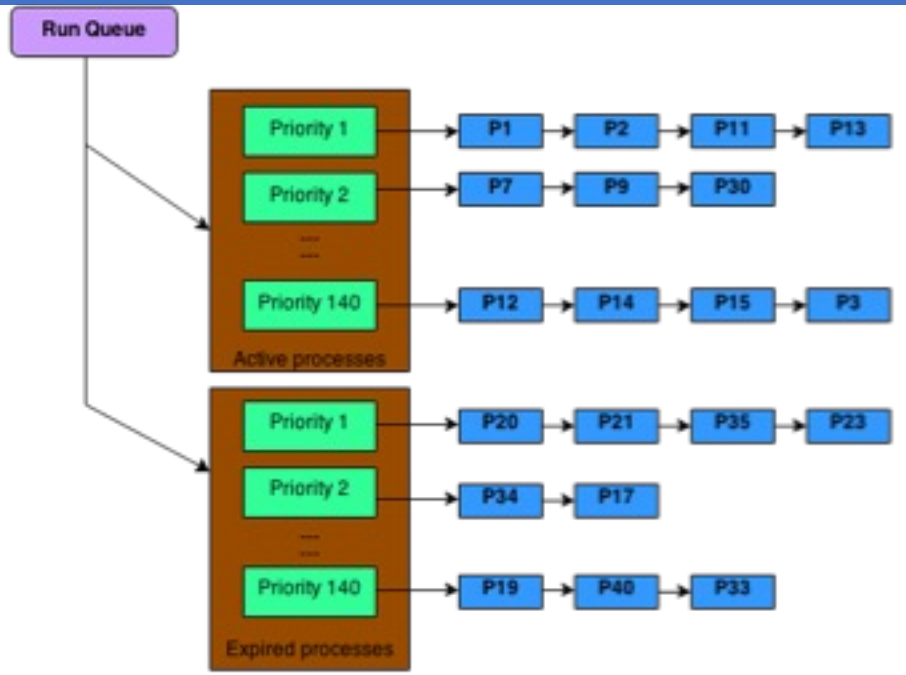


https://www.kernel.org/doc/html/latest/scheduler/sched-nice-design.html

Linux O(1) scheduler

- Preemptive kernel, not scanning all runnable processes
- two queues
  - active queue - stores tasks that have remaining time qu antum
  - expired queue - stores tasks that consumes all its quant um
  - when active is empty, they are switched
- 140 static priorities in each queue
  - different Baseline time slice for different priority





$$\text{Base time slice (in milliseconds)} = \begin{cases} (140 - \text{static priority}) \times 20 & \text{if static priority} < 120 \\ (140 - \text{static priority}) \times 5 & \text{if static priority} \geq 120 \end{cases}$$

Some more details (calibration)

- Dynamic priority
  - make interactive process, which sleeps frequently, have slightly higher priority
  - measure sleep time of a proc.
    - bonus value -5~+5 per 100 ms sleep time
  - keep the interactive proc. in the active queue (not moving to expired queue)
  - if (interactive delta >= bonus) (heuristic value) it means interactive proc.

$$\text{dynamic priority} = \max(100, \min(\text{static priority} - \text{bonus}, 139))$$

$$\text{interactive delta} = (\text{static priority} / 4) - 28$$

Linux CFS (completely fair sched.)

- difficult to distinguish interactive proc. by heuristics
- CFS for performance *and* fairness
- maintain virtual runtime
  - virtual CPU usage (considering the weighted value)
  - scheduler picks the next task based on this value
  - smaller v_runtime means less CPU utilization, compared with the others
    - pick it so that it guarantees fair cpu utilization
- sleeper fairness
  - calibrate the sleep time when calculating the virtual runtime

Time slice in CFS

- Target Latency
  - minimum execution time for once get dispatched by the scheduler
- weight
  - task's weighted cpu share (pre-calculated in table)
  - nice value changed into weight

Nice Value	Weight
-5	3121
-1	1277
0	1024
1	820
5	335

- Time slice of proc i
  = (target latency) x (weight_i) /
  (sum of all weights)
- example
  - TargetLatency = 20 ms, two tasks A, B with nice 0, W_A, W_B = 1024
  - A,B의 CPU share = 1024 / (1024+1024) = 0.5
    - Time slice = 20 * 0.5 = 10 ms

CFS: Task priority adjust

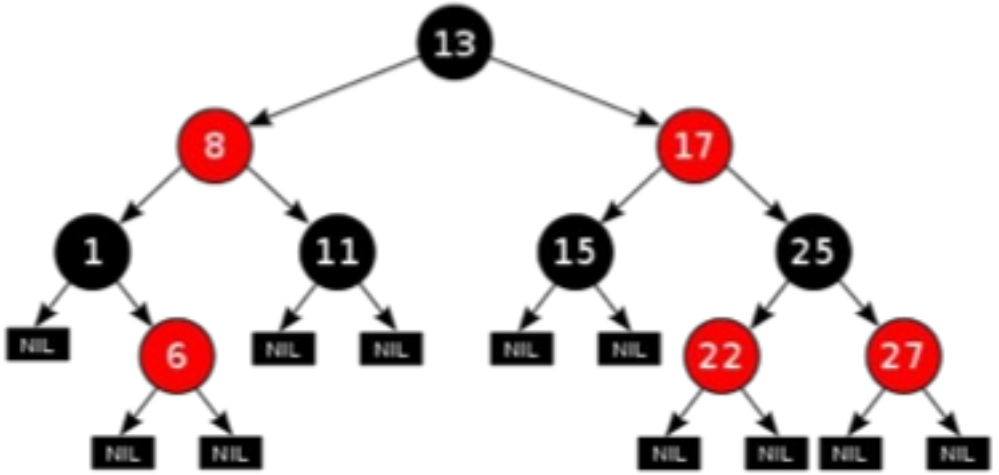
- CFS picks the smallest vruntime
- vruntime considers decaying factor
  - vruntime = t x (decaying factor), where t is cpu time (wall time)
  - decaying factor (weight_0 / weight_i): cpu consumption by nice value, pre-computed in table
  - weight_0 is the default weight
  - higher weighted task has small decay factor, small vruntime, would have higher chance to get scheduled early

Nice Value	Decay Factor
-5	1024/3121 = .33
-1	1024/1277 = .80
0	1024/1024 = 1
1	1024/820 = 1.24
5	1024/335 = 3.05

- nice value
  - increase nice value by 1 makes 10%p less utilization
  - decrease nice value by 1 makes 10%p more utilization
- avoid starvation
  - because vruntime of not-executing tasks becomes smaller

CFS: picking the next proc.

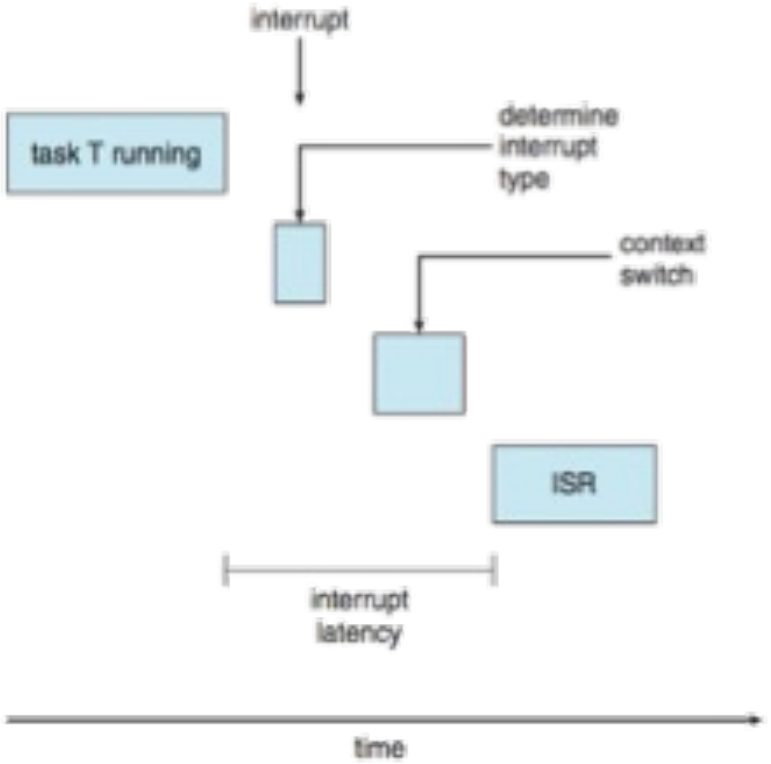
- pick the proc. that has the smallest vruntime
- using the rbtree
  - pick the left-most one,
    which always has the smallest vruntime
  - rebalancing required



- after the execution,
  - measure the execution time in ns
  - re-calculate vruntime
  - if runnable, put it in the rbtree

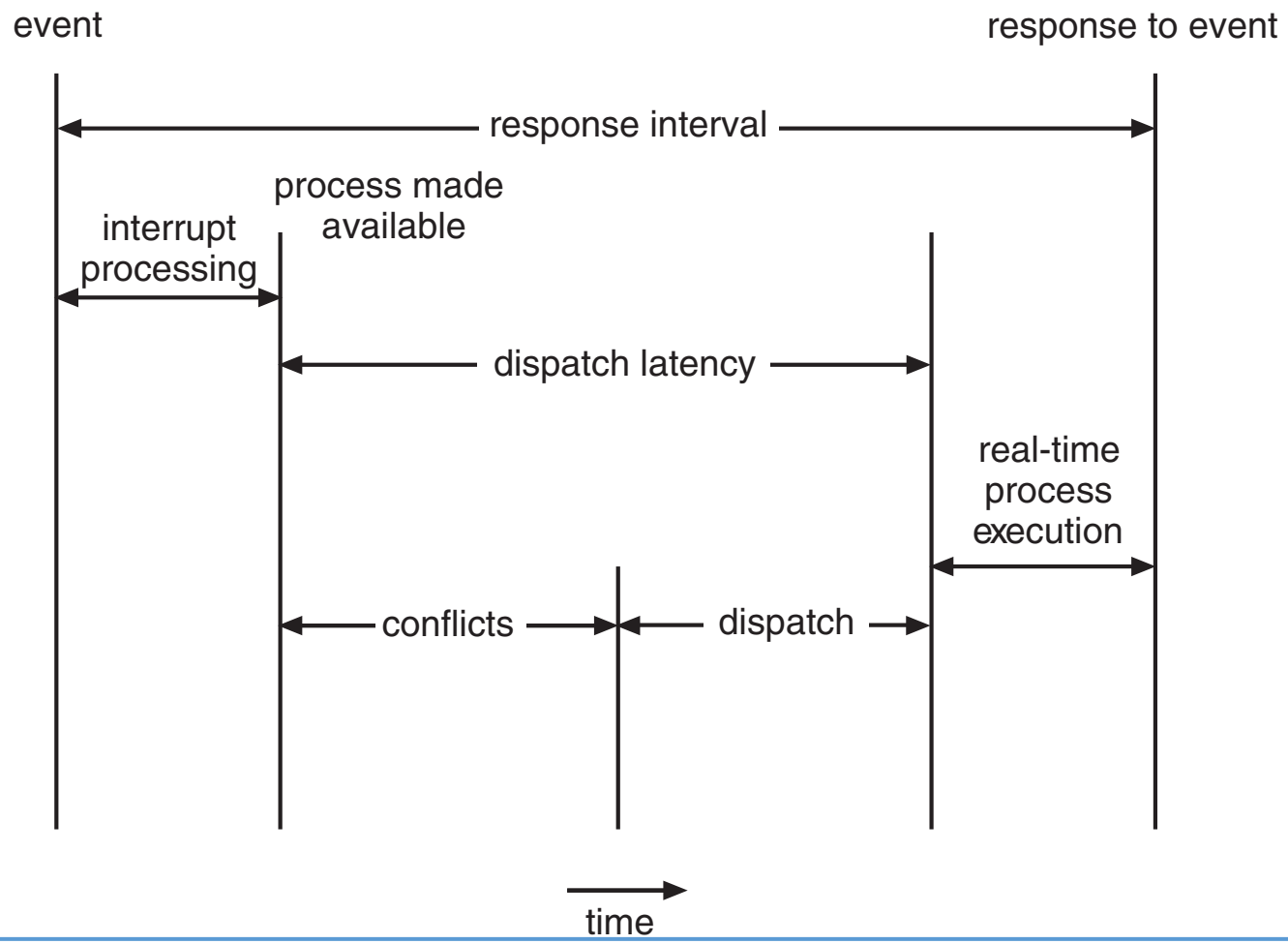
Real-Time CPU Scheduling

- Soft real-time systems
  - requires real-time performance; deadline failure might not be a disaster
  - e.g. streaming player
- Hard real-time systems
  - mission-critical deadline specific systems
  - e.g. control of missiles or rockets
- components in real-time event handling
  - Interrupt latency
    - time duration from Interrupt occurrence to handle it
  - Scheduling dispatch latency
    - time duration from the task becomes ready to be effectively dispatched by the scheduler



Real-Time CPU Scheduling (Cont.)

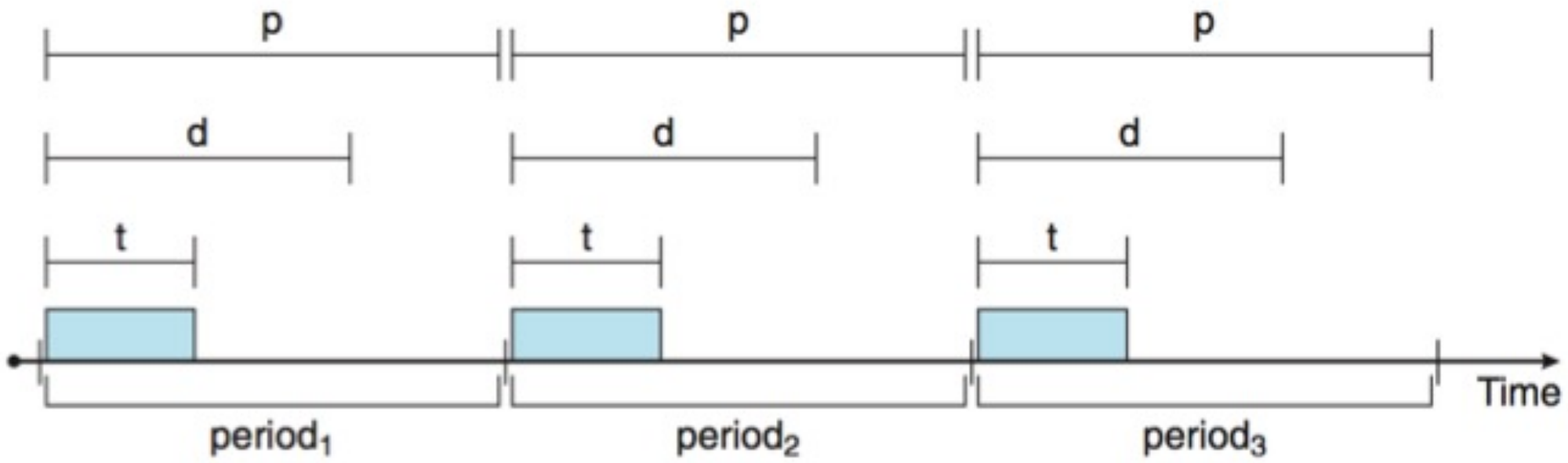
- Conflicts
  - additional latency to dispatch the thread
  - preemption of another thre ad (in-kernel)
  - lock-holder's execution



Priority-based Scheduling

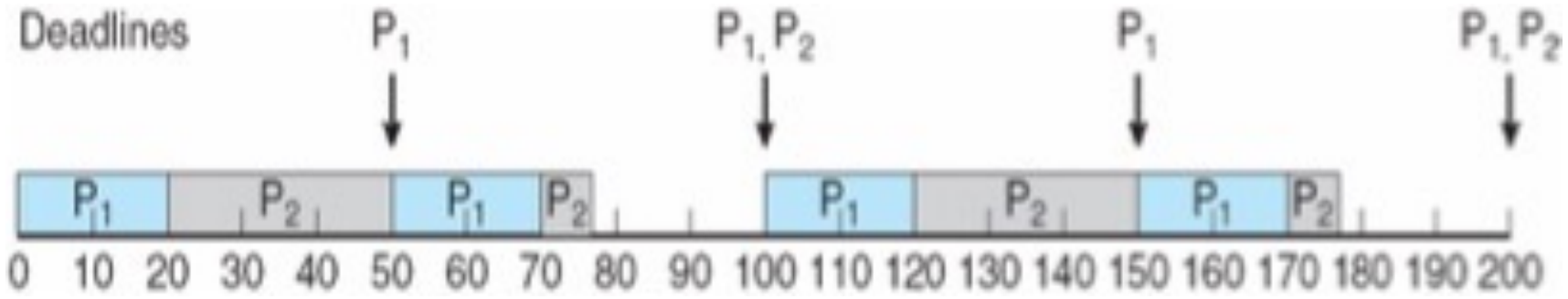
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics:
  periodical execution within a deadline
  - Given processing time t, deadline d, period p
  - $0 \leq t \leq d \leq p$
  - Rate of periodic task is 1/p

Priority-based Scheduling with periodic tasks



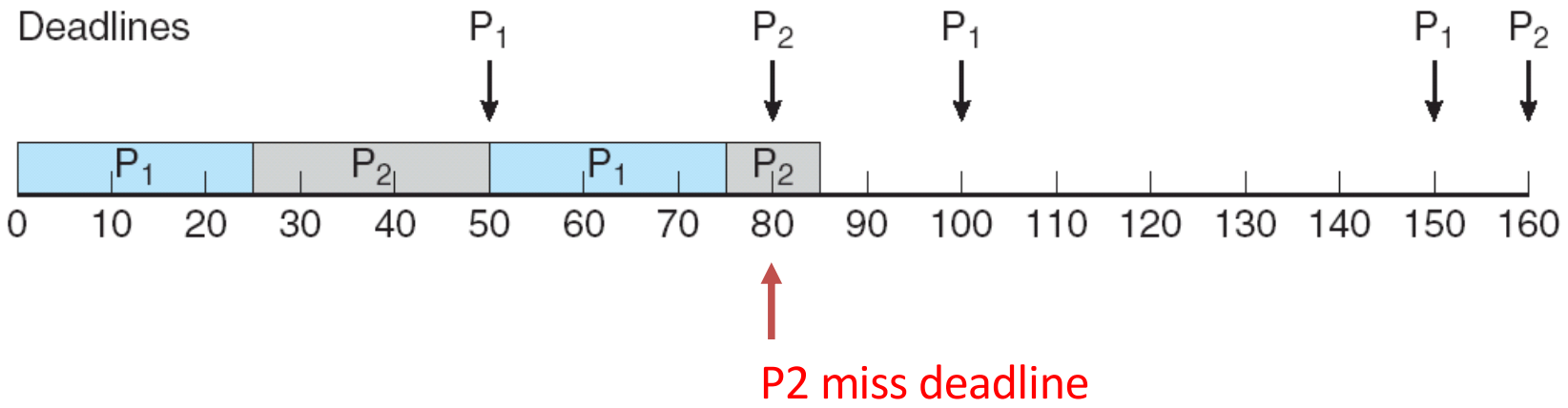
Rate Monotonic (RM) Scheduling

- A priority is assigned based on the rate of periodic task (1/p)
  - Shorter periods = higher priority;
  - Longer periods = lower priority
- P1 is assigned a higher priority than P2.
  - P1: p and d = 50, t = 20 → high priority (1/50)
  - P2: p and d = 100, t= 35 → low priority (1/100)



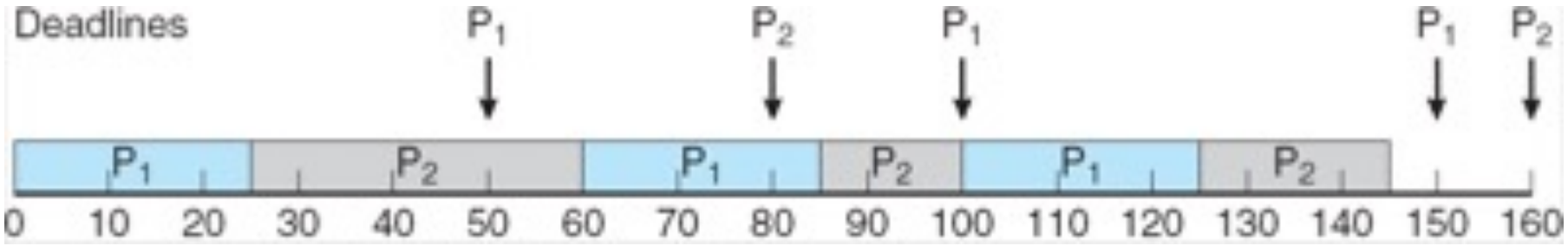
Missed Deadlines with Rate Monotonic Scheduling

- Not suitable for hard real-time system
  - P1: p and d = 50, t = 25
  - P2: p and d = 80, t = 35



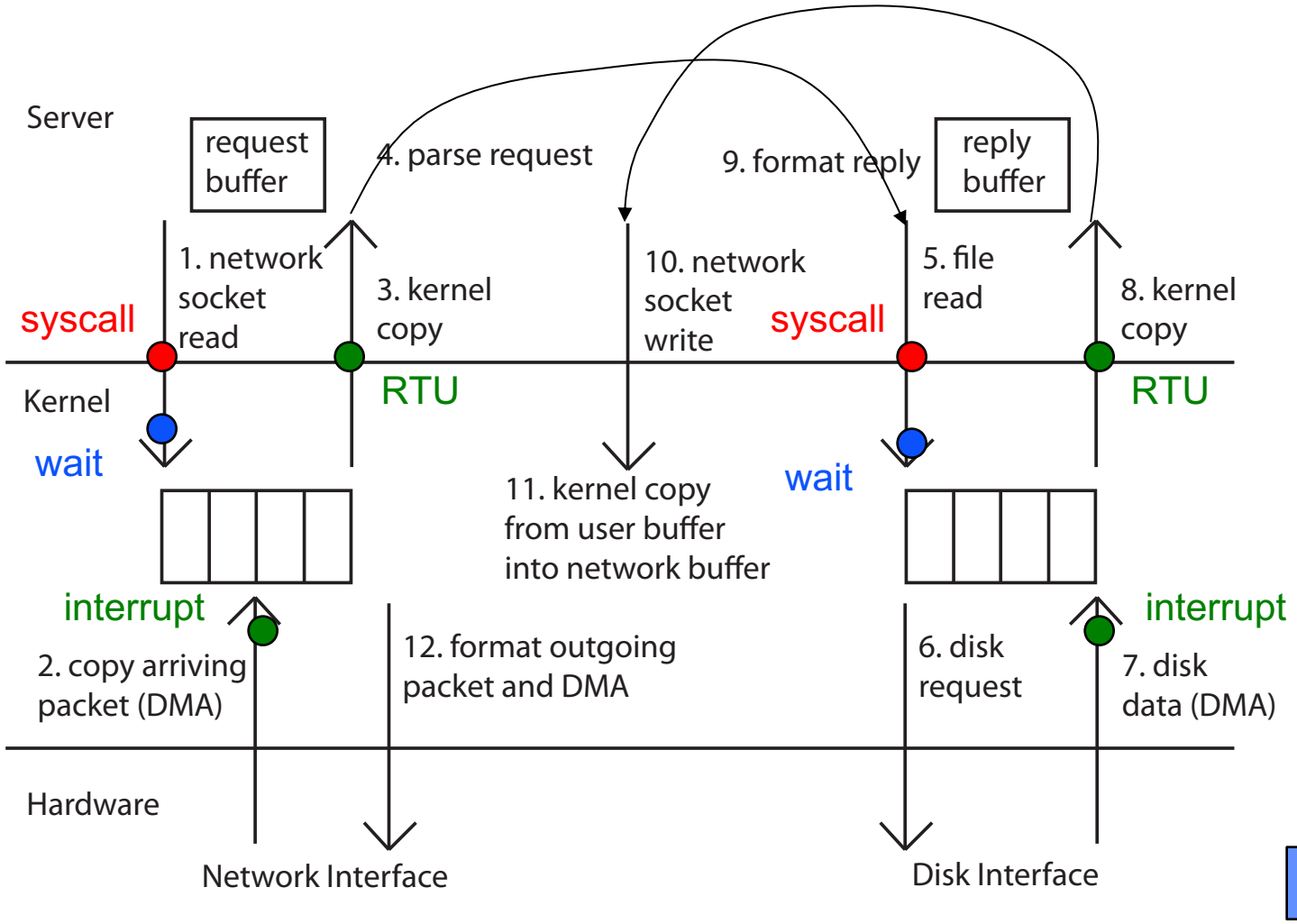
Earliest Deadline First (EDF) Scheduling (dynamic scheduling)

- Priorities are assigned according to deadlines:
  - the earlier the deadline, the higher the priority;
  - the later the deadline, the lower the priority



- P1: p and d = 50, t = 25
- P2: p and d = 80, t= 35

Putting it together: web server



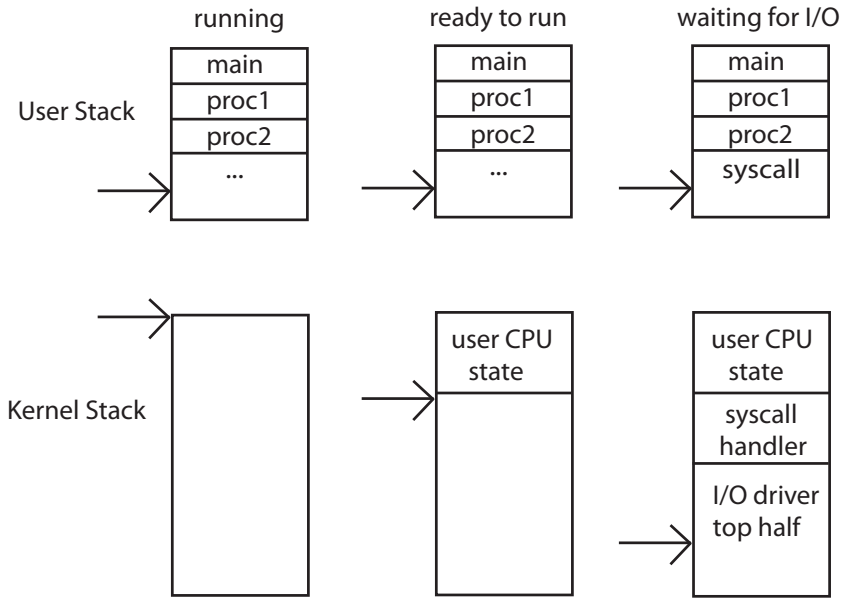
Digging Deeper: Discussion & Questions

Implementing Safe Mode Transfers

- Carefully constructed kernel code packs up the user process state an sets it aside.
  - Details depend on the machine architecture
- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself.
- Interrupt processing must not be visible to the user process (why?)
  - Occurs between instructions, restarted transparently
  - No change to process state
  - What can be observed even with perfect interrupt processing?

Kernel Stack Challenge

- Kernel needs space to work
- Cannot put anything on the user stack (Why?)
- Two-stack model
  - OS thread has interrupt stack (located in kernel memory) plus User stack (located in user memory)
  - Syscall handler copies user args to kernel space before invoking specific function (e.g., open)
  - Interrupts (???)



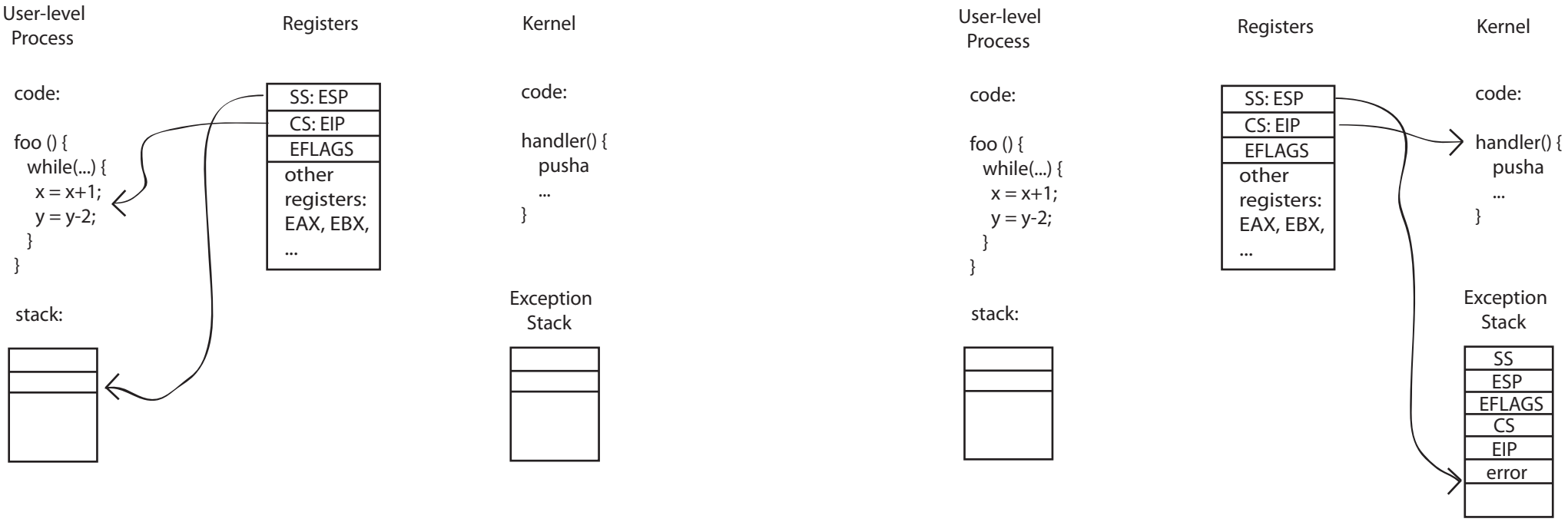
Hardware support: Interrupt Control

- Interrupt Handler invoked with interrupts 'disabled'
  - Re-enabled upon completion
  - Non-blocking (run to completion, no waits)
  - Pack it up in a queue and pass off to an OS thread to do the hard work
    - wake up an existing OS thread
- OS kernel may enable/disable interrupts
  - On x86: CLI (disable interrupts), STI (enable)
  - Atomic section when select next process/thread to run
  - Atomic return from interrupt or syscall
- HW may have multiple levels of interrupt
  - Mask off (disable) certain interrupts, eg., lower priority
  - Certain non-maskable-interrupts (nmi)
    - e.g., kernel segmentation fault

How do we take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel
- Kernel interrupt stack
  - Handler works regardless of state of user code
- Interrupt masking
  - Handler is non-blocking
- Atomic transfer of control
  - “Single instruction”-like to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred

Before & during



Kernel System Call Handler

- Locate arguments
  - In registers or on user(!) stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - into user memory

Multiprocessors - Multicores – Multiple Threads

- What do we need to support Multiple Threads
  - Multiple kernel threads?
  - Multiple user threads in a process?
- What if we have multiple Processors / Cores

Idle Loop & Power

- Measly do-nothing unappreciated trivial piece of code that is central to low-power

Performance

- Performance = Operations / Time

- How can the OS ruin application performance?
- What can the OS do to increase application performance?

4 OS concepts working together

- Privilege/User Mode
  - The hardware can operate in two modes, with only the “system” mode having the ability to access certain resources.
- Address Space
  - Programs execute in an *address space* that is distinct from the memory space of the physical machine
- Process
  - An instance of an executing program is *a process consisting of an address space and one or more threads of control*
- Protection
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses

break Question

- Process is an instance of a program executing.
  - The fundamental OS responsibility
- Processes do their work by processing and calling file system operations
- Are their any operations on processes themselves?
- exit ?

Programming Interface with process

System call & library functions

- application programming interface (API)
  - language-specific utility functions to do some tasks
  - c.f.) application binary interface
    - binary interface vs. programming interface
    - application vs. operating system (system)
  - c.f.) instruction set architecture
    - instructions with registers
- system call is a binary interface
  - some CPUs have ‘syscall’ instruction
  - also, we need a programming interface
  - we can interact with OS using syscall programming interface

processes in the OS, checking the pid

- Try ps, ps -ef
- check pid of bash
- and you can make a small program that prints pid of the process

pid.c


```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024
int main(int argc, char *argv[])
{
  int c;

  pid_t pid = getpid();   /* get current processes PID */

  printf("My pid: %d\n", pid);

  c = fgetc(stdin);
  exit(0);
}
```

processes in the OS, checking the pid

- Try ps, ps -ef
- check pid of bash
- and you can make a small program that prints pid of the process
- and you can make another process from it
  - fork() syscall creates a copy of the process

forking a process


```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024
int main(int argc, char *argv[])
{
  char buf[BUFSIZE];
  size_t readlen, writelen, slen;
  pid_t cpid, mypid;
  pid_t pid = getpid();          /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  }  else if (cpid == 0) {       /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
    exit(1);
  }
  exit(0);
}
```

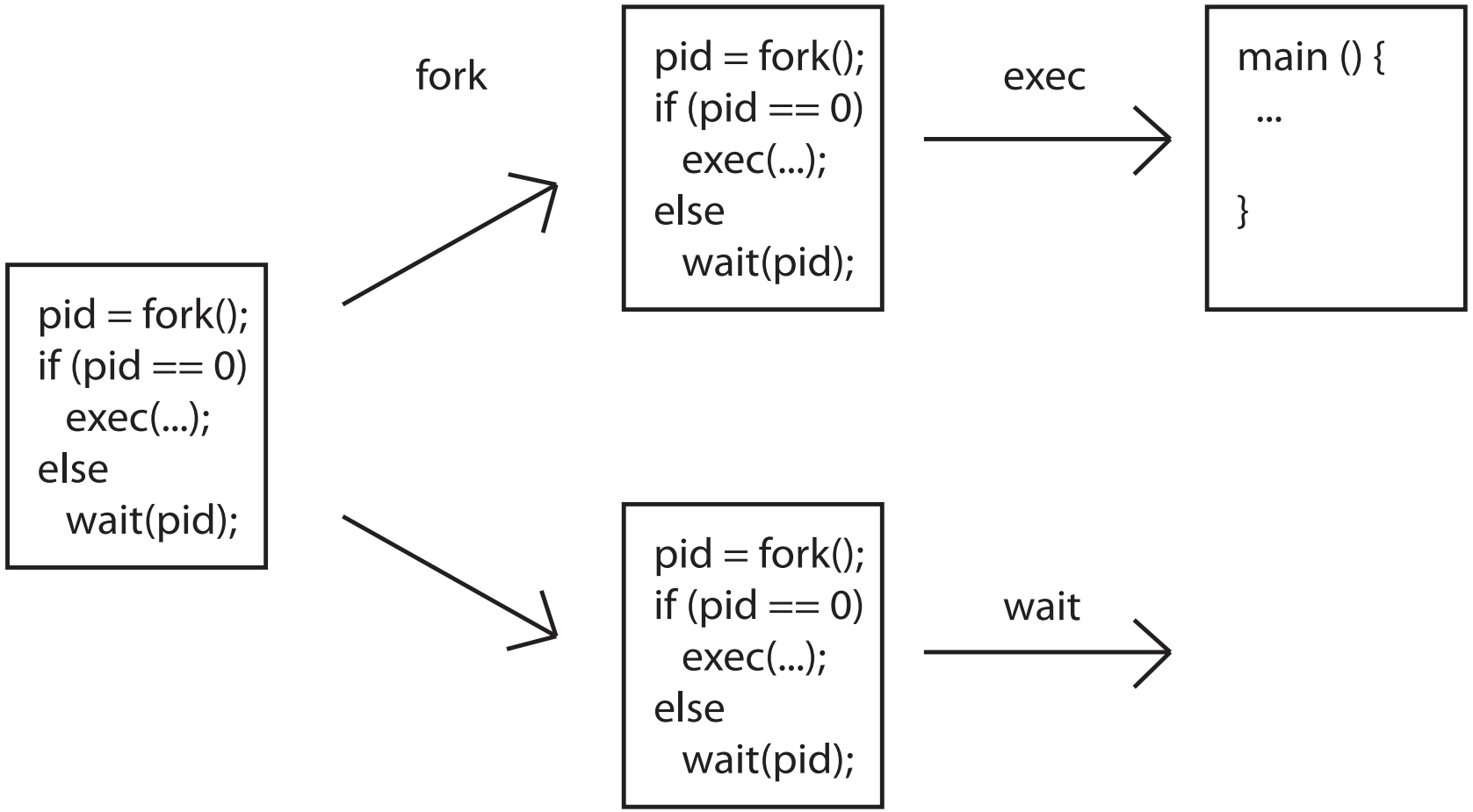
UNIX Process Management

- UNIX fork - system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec - system call to *change the program* being run by the current process
- UNIX wait - system call to wait for a process to finish
- UNIX signal - system call to send a notification to another process

fork v2


```
…
cpid = fork();
if (cpid > 0) {                    /* Parent Process */
  mypid = getpid();
  printf("[%d] parent of [%d]\n", mypid, cpid);
  tcpid = wait(&status);
  printf("[%d] bye %d\n", mypid, tcpid);
}  else if (cpid == 0) {           /* Child Process */
  mypid = getpid();
  printf("[%d] child\n", mypid);
}
…
```

UNIX Process Management



Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells

- Example: to compile a C program

```
cc -c sourcefile1.c
cc -c sourcefile2.c
ln -o program sourcefile1.o sourcefile2.o
./program
```

Signals – infloop.c


```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>

#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum)
{
  printf("Caught signal %d - phew!\n",signum);
  exit(1);
}

int main() {
  signal(SIGINT, signal_callback_handler);

  while (1) {}
}
```

Process races: fork.c


```
if (cpid > 0) {
   mypid = getpid();
   printf("[%d] parent of [%d]\n", mypid, cpid);
   for (i=0; i<100; i++) {
     printf("[%d] parent: %d\n", mypid, i);
     //       sleep(1); }
 }  else if (cpid == 0) {
   mypid = getpid();
   printf("[%d] child\n", mypid);
   for (i=0; i>-100; i--) {
     printf("[%d] child: %d\n", mypid, i);
     //       sleep(1);
   }
 }
```

BIG OS Concepts so far

- Processes
- Address Space
- Protection
- Dual Mode
- Interrupt handlers (including syscall and trap)
- File System
  - Integrates processes, users, cwd, protection
- Key Layers: OS Lib, Syscall, Subsystem, Driver
  - User handler on OS descriptors
- Process control
  - fork, wait, signal --- exec

Design Considerations for old OS

- Functions at different layers
  - Where to locate which?
    - concepts around OS Design: correctness, efficiency, fault-tolerant
    - Butler W. Lampson and Robert F. Sproull. 1979. An open operating system for a single-user machine. In *Proceedings of the seventh ACM symposium on Operating systems principles* (SOSP '79).
    - Butler W. Lampson. 1983. Hints for computer system design. In *Proceedings of the ninth ACM symposium on Operating systems principles* (SOSP '83)
    - J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-end arguments in system design. *ACM Trans. Comput. Syst.* 2, 4 (November 1984), 277-288
    - Dennis M. Ritchie and Ken Thompson. 1974. The UNIX time-sharing system. *Commun. ACM* 17, 7 (July 1974), 365-375
    - Fernando J. Corbató. 1991. On building systems that will fail. *Commun. ACM* 34, 9 (September 1991), 72-81.
  - What's good for layering?
    - strictly limit the accessible abstractions

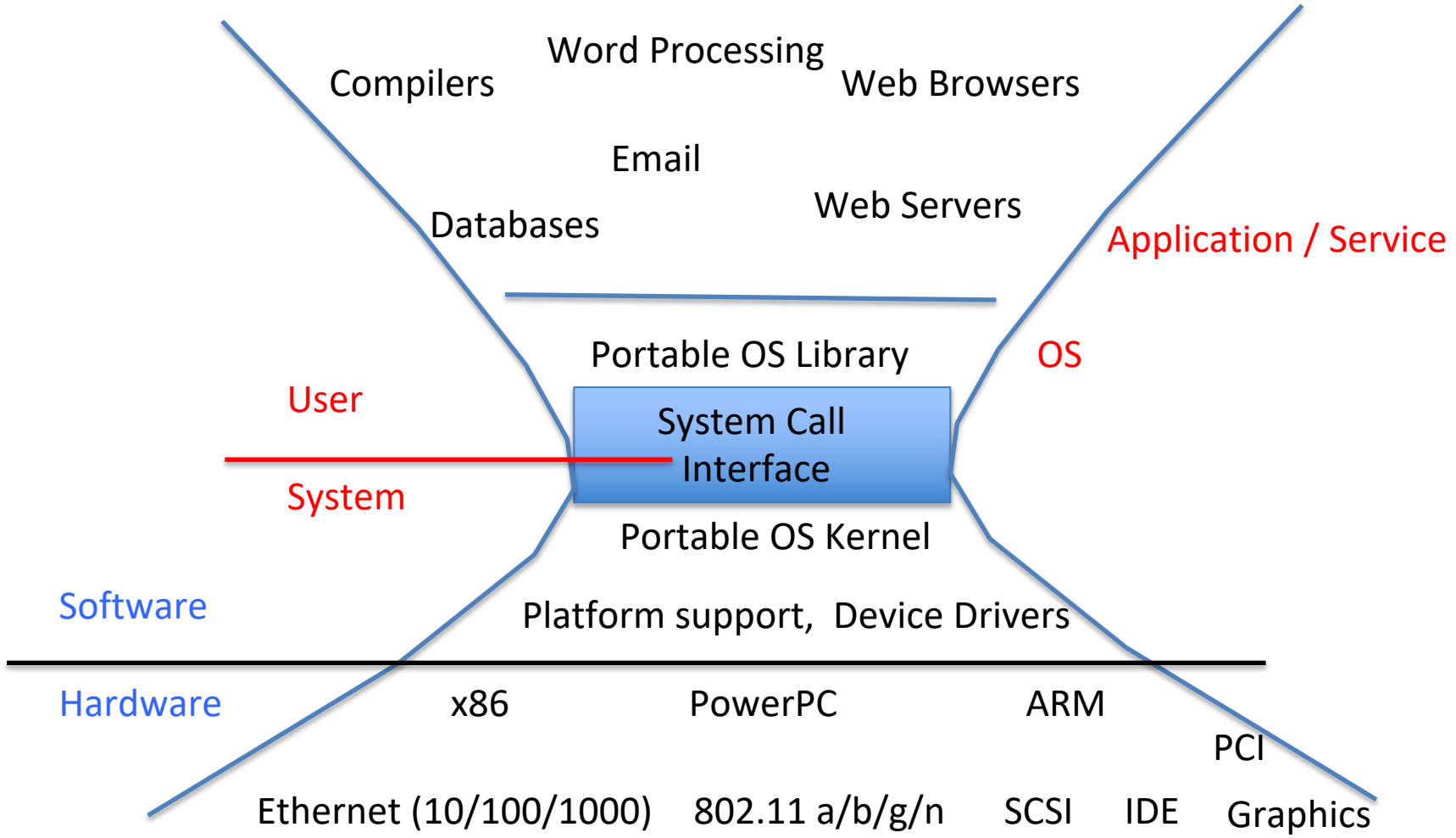
Key Unix I/O Design Concepts

- Uniformity
  - file operations, device I/O, and interprocess communication through open, read/write, clo se
  - Allows simple composition of programs
    - find | grep | wc …
- Open before use
  - Provides opportunity for access control and arbitration
  - Sets up the underlying machinery, i.e., data structures
- Byte-oriented
  - Even if blocks are transferred, addressing is in bytes
- Kernel buffered reads
  - Streaming and block devices looks the same, read blocks yielding processor to other task
- Kernel buffered writes
  - Completion of out-going transfer decoupled from the application, allowing it to continue
- Explicit close

Layering of Software

- When it is good
  - Hide details
    - A user can focus only on what they are doing
  - Least dependency
    - Only the upper layer is dependent upon this
    - clear R&R (Roles and Responsibility)
  - Compatibility & Flexibility
    - Using the same code for different OS
- API (Application Programming Interface) vs.
  ABI (Application Binary Interface)
  - API: language-specific, function prototype,
    public methods of classes
    - that you can call
  - ABI: binary interface between HW and your application, language-independent, OS specific interface

A Kind of Narrow Waist

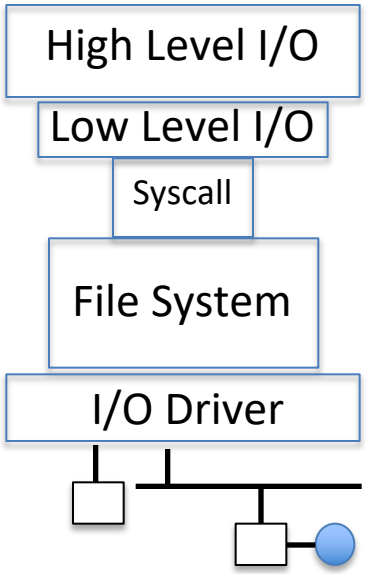


The file abstraction

- File
  - Named collection of data in a file system
  - File data
    - Text, binary, linearized objects
  - File Metadata: information about the file
    - Size, Modification Time, Owner, Security info
    - Basis for access control
- Directory
  - “Folder” containing files & Directories
  - Hierachical (graphical) naming
    - Path through the directory graph
    - Uniquely identifies a file or directory
      - /home/seehwan.yoo/public_html/index.html
  - Links and Volumes (later)

I/O & Storage Layers

Application / Service



streams

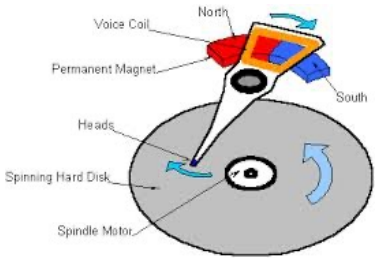
handles

registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA



C high level File API – streams (review)

- Operate on “streams” - sequence of bytes, whether text or data, with a position

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *fp );
```

Mode Text	Binary	Descriptions
r	rb	Open existing file for reading
w	wb	Open for writing; created if does not exist
a	ab	Open for appending; created if does not exist
r+	rb+	Open existing file for reading & writing.
w+	wb+	Open for reading & writing; truncated to zero if exists, create otherwise
a+	ab+	Open for reading & writing. Created if does not exist. Read from beginning, write as append

Connecting Processes, Filesystem, and Users

- Process has a ‘current working directory’
- Absolute Paths (full path)
  - /home/seehwan.yoo/
- Relative paths
  - index.html, ./index.html - current WD
    - ./a.out
  - ../index.html - parent of current WD
  - ~, ~seehwan.yoo - home directory

C API Standard Streams

- Three predefined streams are opened implicitly when the program is executed.
  - `FILE *stdin` - normal source of input, can be redirected
  - `FILE *stdout` - normal source of output, can too
  - `FILE *stderr` - diagnostics and errors

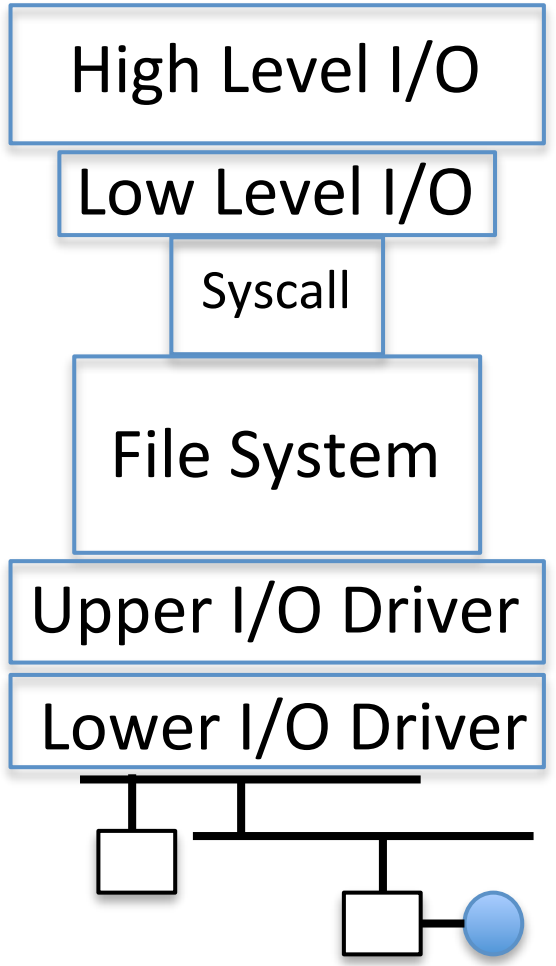
- STDIN / STDOUT enable composition in Unix

C high level File API – stream ops


```
#include <stdio.h>
// character oriented
int fputc( int c, FILE *fp );                          // rtn c or EOF on err
int fputs( const char *s, FILE *fp );          // rtn >0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

// formatted
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int fscanf(FILE *restrict stream, const char *restrict format, ... );
```

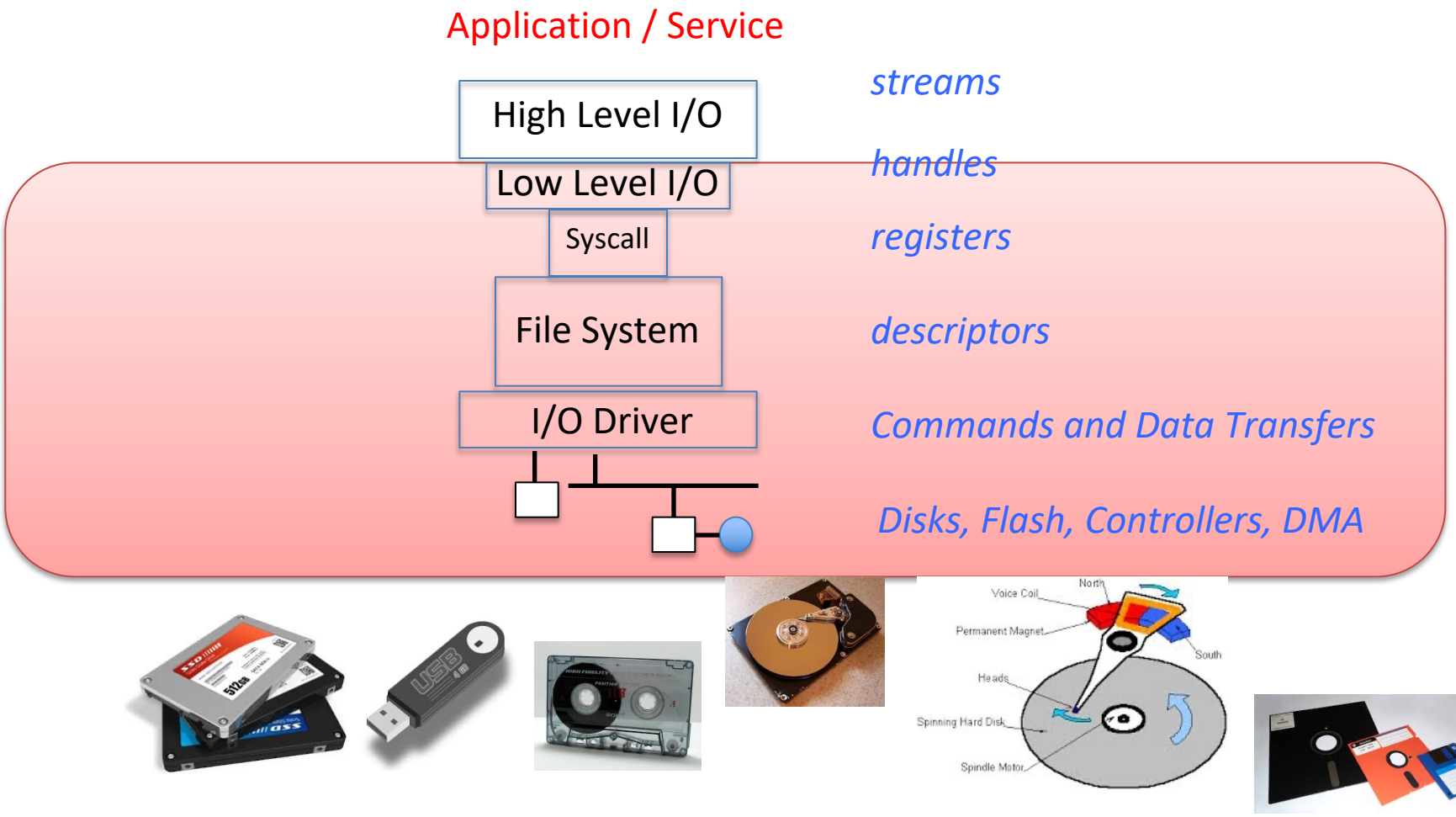
C Stream API positioning


```
int fseek(FILE *stream, long int offset, int whence);
long int ftell (FILE *stream)
void rewind (FILE *stream)
```



- Preserves high level abstraction of a uniform stream of objects

What's below the surface ??



C Low level I/O

- Operations on File Descriptors - as OS object representing the state of a file
  - User has a “handle” on the descriptor

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int open (const char *filename, int flags [, mode_t mode])
int creat (const char *filename, mode_t mode)
int close (int filedes)
```

Bit vector of:
- Access modes (Rd, Wr, …)
- Open Flags (Create, …)
- Operating modes (Appends, …)

Bit vector of Permission Bits:
- User|Group|Other X R|W|X

http://www.gnu.org/software/libc/manual/html_node/Opening-and-Closing-Files.html

C Low Level: standard descriptors


```
#include <unistd.h>

STDIN_FILENO -  macro has value 0
STDOUT_FILENO - macro has value 1
STDERR_FILENO - macro has value 2

int fileno (FILE *stream)

FILE * fdopen (int filedes, const char *opentype)
```

- Crossing levels: File descriptors vs. streams
- Don’t mix them!

C Low Level Operations


```
ssize_t read (int filedes, void *buffer, size_t maxsize)
 - returns bytes read, 0 => EOF, -1 => error
ssize_t write (int filedes, const void *buffer, size_t size)
 - returns bytes written

off_t lseek (int filedes, off_t offset, int whence)

int fsync (int fildes) – wait for i/o to finish
void sync (void) – wait for ALL to finish
```

- When write returns,
  data is on its way to disk and can be read,
  but it may not actually be permanent!

A little example: lowio.c


```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
  char buf[1000];
  int     fd = open("lowio.c", O_RDONLY, S_IRUSR | S_IWUSR);
  ssize_t rd = read(fd, buf, sizeof(buf));
  int    err = close(fd);
  ssize_t wr = write(STDOUT_FILENO, buf, rd);
}
```

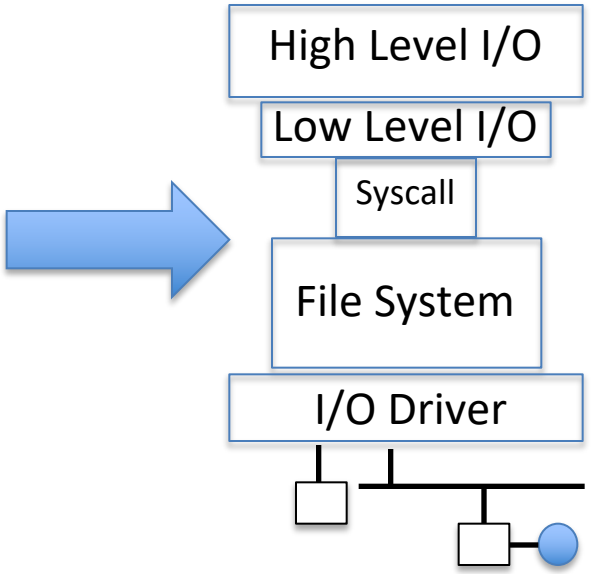
And lots more !

- TTYs versus files
- Memory mapped files
- File Locking
- Asynchronous I/O
- Generic I/O Control Operations
- Duplicating descriptors
- man pages
- The Linux Programming Interfaces - https://man7.org/tlpi/

```
int dup2 (int old, int new)
int dup (int old)
```

What's below the surface ??

Application / Service



streams

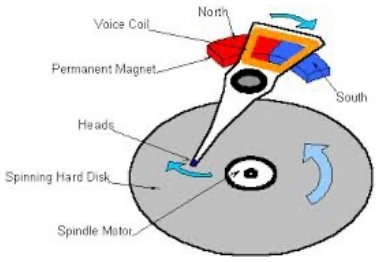
handles

registers

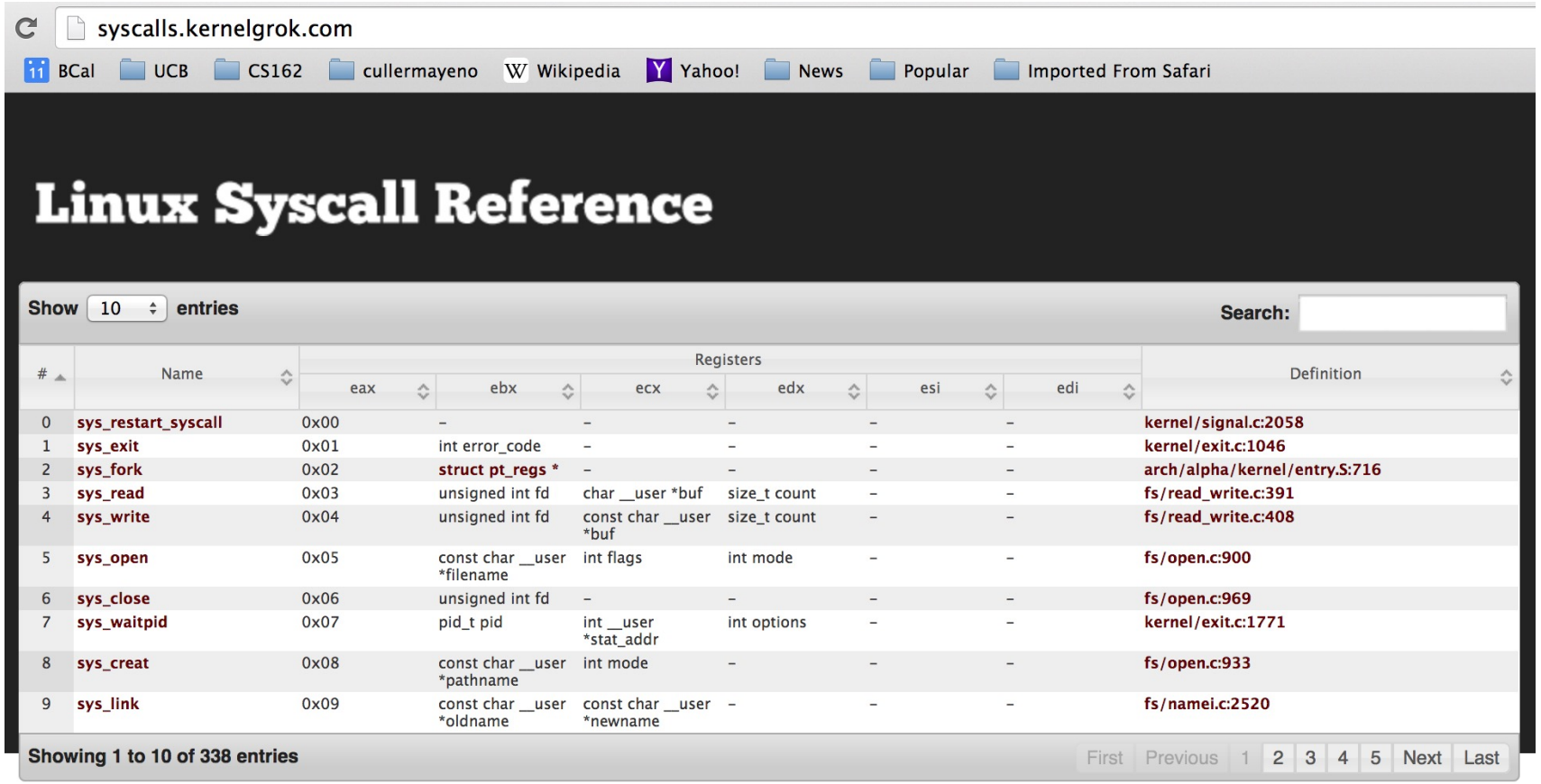
descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA



Syscall



syscalls.kernelgrok.com

BCal | UCB | CS162 | cullermayeno | Wikipedia | Yahoo! | News | Popular | Imported From Safari

Linux Syscall Reference

Show 10 entries Search:

#	Name	Registers						Definition
		eax	ebx	ecx	edx	esi	edi	
0	sys_restart_syscall	0x00	-	-	-	-	-	kernel/signal.c:2058
1	sys_exit	0x01	int error_code	-	-	-	-	kernel/exit.c:1046
2	sys_fork	0x02	struct pt_regs *	-	-	-	-	arch/alpha/kernel/entry.S:716
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count	-	-	fs/read_write.c:391
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	fs/read_write.c:408
5	sys_open	0x05	const char __user *filename	int flags	int mode	-	-	fs/open.c:900
6	sys_close	0x06	unsigned int fd	-	-	-	-	fs/open.c:969
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options	-	-	kernel/exit.c:1771
8	sys_creat	0x08	const char __user *pathname	int mode	-	-	-	fs/open.c:933
9	sys_link	0x09	const char __user *oldname	const char __user *newname	-	-	-	fs/namei.c:2520

Showing 1 to 10 of 338 entries First Previous 1 2 3 4 5 Next Last

Generated from Linux kernel 2.6.35.4 using Exuberant Ctags, Python, and DataTables.
Project on GitHub. Hosted on GitHub Pages.

- Low level lib parameters are set up in registers and syscall instruction is issued

Internal OS 'File Descriptor'

- Internal data structure describing everything about the file
  - Where it resides
  - Its status
  - How to access it

lxr.free-electrons.com/source/include/linux/fs.h#L747

BCal UCB CS162 cullermayeno Wikipedia Yahoo! News

```
746
747 struct file {
748         union {
749                 struct llist_node       fu_llist;
750                 struct rcu_head         fu_rcuhead;
751         } f_u;
752         struct path             f_path;
753 #define f_dentry        f_path.dentry
754         struct inode            *f_inode;       /* cach
755         const struct file_operations    *f_op;
756
757         /*
758          * Protects f_ep_links, f_flags.
759          * Must not be taken from IRQ context.
760          */
761         spinlock_t              f_lock;
762         atomic_long_t           f_count;
763         unsigned int            f_flags;
764         fmode_t                 f_mode;
765         struct mutex            f_pos_lock;
766         loff_t                  f_pos;
767         struct fown_struct      f_owner;
768         const struct cred       *f_cred;
769         struct file_ra_state    f_ra;
770
771         u64                     f_version;
772 #ifdef CONFIG_SECURITY
773         void                    *f_security;
774 #endif
775         /* needed for tty driver, and maybe others */
776         void                    *private_data;
777
778 #ifdef CONFIG_EPOLL
779         /* Used by fs/eventpoll.c to link all the hooks
780         struct list_head        f_ep_links;
781         struct list_head        f_tfile_llink;
782 #endif /* #ifdef CONFIG_EPOLL */
783         struct address_space    *f_mapping;
784 } __attribute__((aligned(4)));  /* lest something weir
```

File System: from syscall to driver

In fs/read_write.c

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```

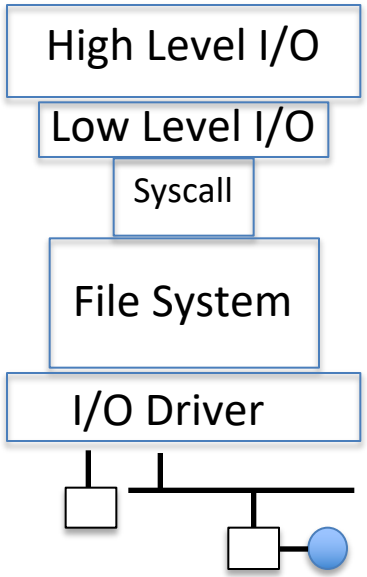
Low Level Driver

- Associated with particular hardware device
- Registers / Unregisters itself with the kernel
- Handler functions for each of the file operations

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    [...]
};
```

So what happens when you fgetc?

Application / Service



streams

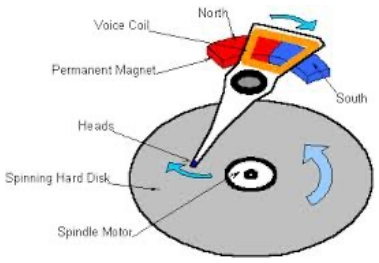
handles

registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA



Summary

- Four Concepts in Process
  - Process
  - Dual mode execution
  - Protection
  - Address space
- Unix Design and software layers
  - Programming interface
  - high-level abstract API, low-level C lib, syscalls

Mini-lab.