

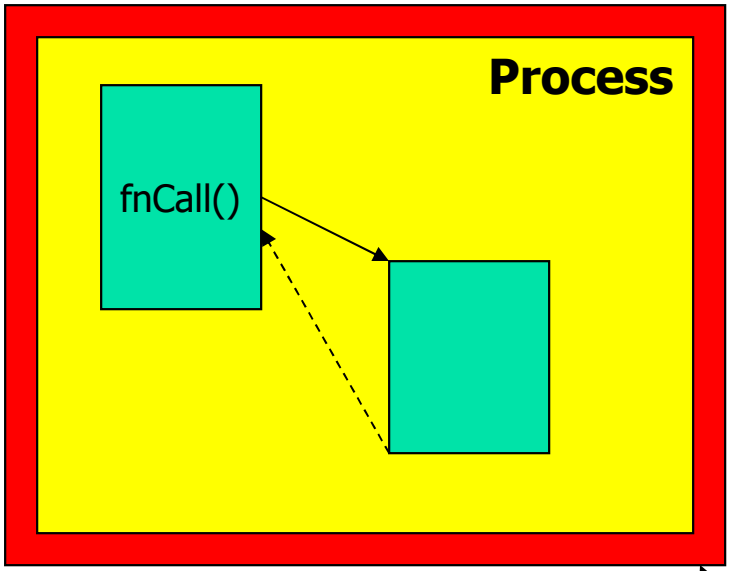
# Midterm Review
# Operating System Design

University of Illinois at Urbana-Champaign

CS 423

# System Calls

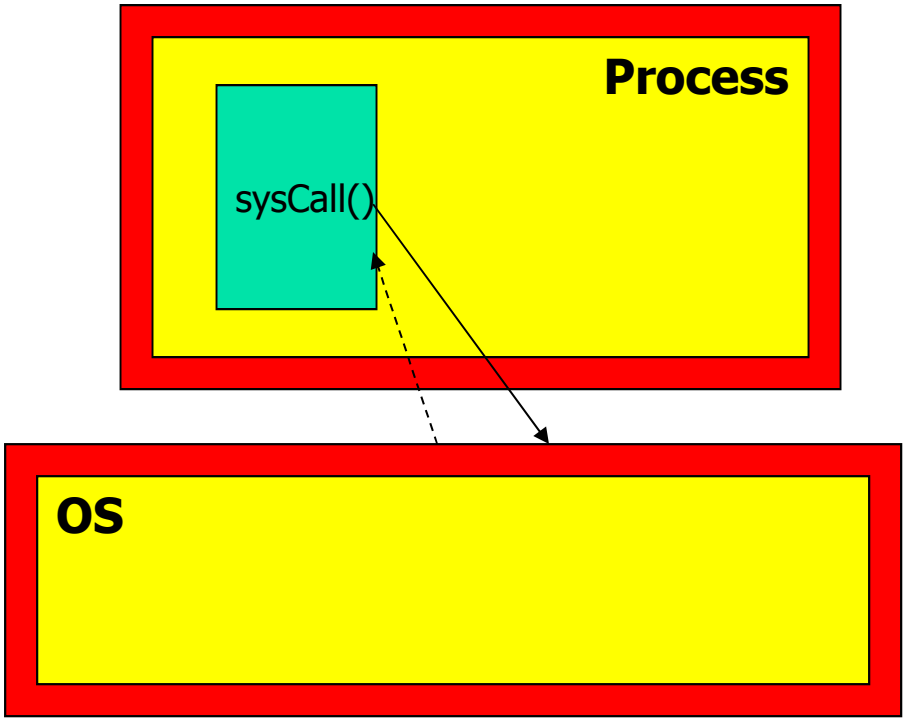
## Function Call



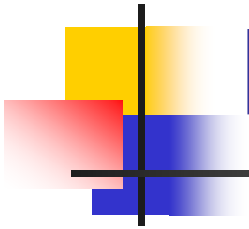

Caller and callee are in the same Process

- Same user
- Same "domain of trust"

## System Call




- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse



# Examples of System Calls

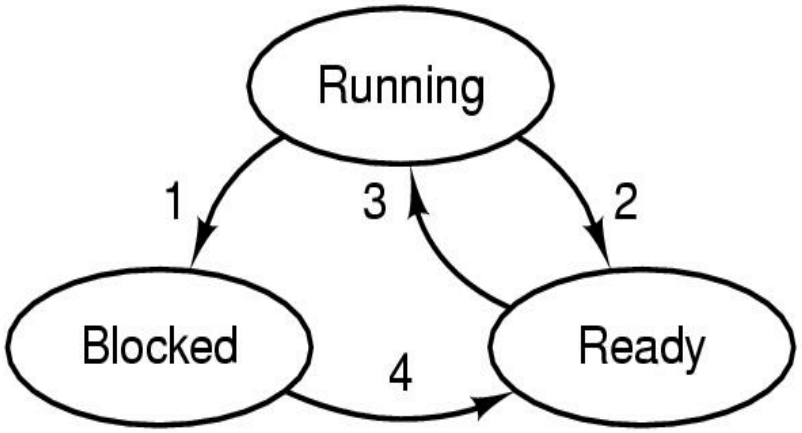
- Example:
  - getuid()  //get the user ID
  - fork()  //create a child process
  - exec()  //executing a program
- Don't mix system calls with standard library calls
  - Differences?
  - Is printf() a system call?
  - Is rand() a system call?

# I/O Library Calls versus System Calls

- **Each system call has analogous procedure calls from the standard I/O library:**

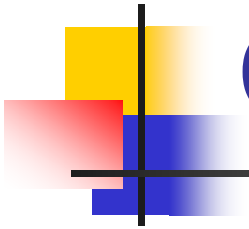
| System Call | Standard I/O call |
|---|---|
| open | fopen |
| close | fclose |
| read/write | getchar/putchar |
| | getc/putc |
| | fgetc/fputc |
| | fread/fwrite |
| | gets/puts |
| | fgets/fputs |
| | scanf/printf |
| | fscanf/fprintf |
| lseek | fseek |

# Processes



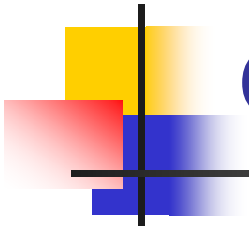

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process states
  - Running (occupy CPU)
  - Blocked
  - Ready (does not occupy CPU)
  - Other states: suspended, terminated
- Question: in a single processor machine, how many process can be in running state?



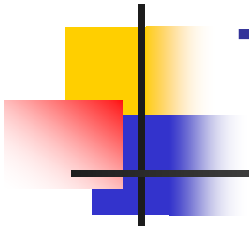
# Creating a Process – fork()

- fork() duplicates a process so that instead of one process you get two.
  - The new process and the old process both continue in parallel from the statement that follows the fork()
- fork() returns
  - 0 if child
  - -1 if fork fails
  - Child's PID if parent process
- Child gets new program counter, stack, file descriptors, heap, globals, pid!



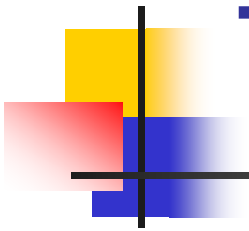
# exec() Function

- Exec function allows child process to execute code that is different from that of parent
- Exec family of functions provides a facility for overlaying the process image of the calling process with a new image.
- Exec functions return -1 and sets errno if unsuccessful



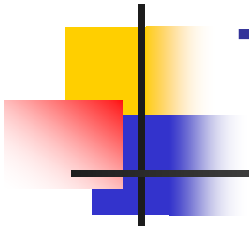
# Threads and Processes

| **Per process items** | **Per thread items** |
|---|---|
| Address space | Program counter |
| Global variables | Registers |
| Open files | Stack |
| Child processes | State |
| Pending alarms | |
| Signals and signal handlers | |
| Accounting information | |



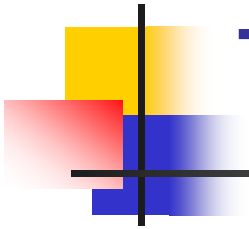
# Threads

| POSIX function | description |
|---|---|
| pthread_create | create a thread |
| pthread_detach | set thread to release resources |
| pthread_equal | test two thread IDs for equality |
| pthread_exit | exit a thread without exiting process |
| pthread_kill | send a signal to a thread |
| pthread_join | wait for a thread |
| pthread_self | find out own thread ID |



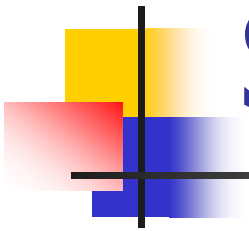
# Kernel versus User Threads: Trade-offs?

- Kernel thread packages
  - Each thread can make blocking I/O calls
  - Can run concurrently on multiple processors
- Threads in User-level
  - Fast context switch
  - Customized scheduling



# Things Suitable for Threading

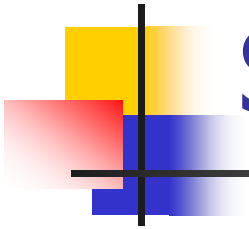
- Block for potentially long waits
- Use many CPU cycles
- Respond to asynchronous events
- Execute functions of different importance
- Execute parallel code



# Signals

- Signal is *generated* when the event that causes it occurs.
- Signal is *delivered* when a process receives it.
- The *lifetime* of a signal is the interval between its generation and delivery.
- Signal that is generated but not delivered is *pending*.
- Process *catches* signal if it executes a *signal handler* when the signal is delivered.
- Alternatively, a process can *ignore* a signal when it is delivered, that is to take no action.
- Process can temporarily prevent signal from being delivered by *blocking* it.
- *Signal Mask* contains the set of signals currently blocked.

# Examples of POSIX Required Signals

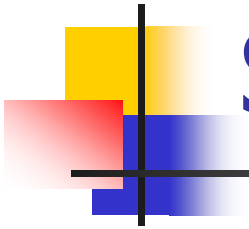


| Signal | Description | default action |
|---|---|---|
| SIGABRT | process abort | implementation dependent |
| **SIGALRM** | **alarm clock** | **abnormal termination** |
| SIGBUS | access undefined part of memory object | implementation dependent |
| SIGCHLD | child terminated, stopped or continued | ignore |
| SIGILL | invalid hardware instruction | implementation dependent |
| **SIGINT** | **interactive attention signal (usually ctrl-C)** | **abnormal termination** |
| **SIGKILL** | **terminated (cannot be caught or ignored)** | **abnormal termination** |

# Examples of POSIX Required Signals

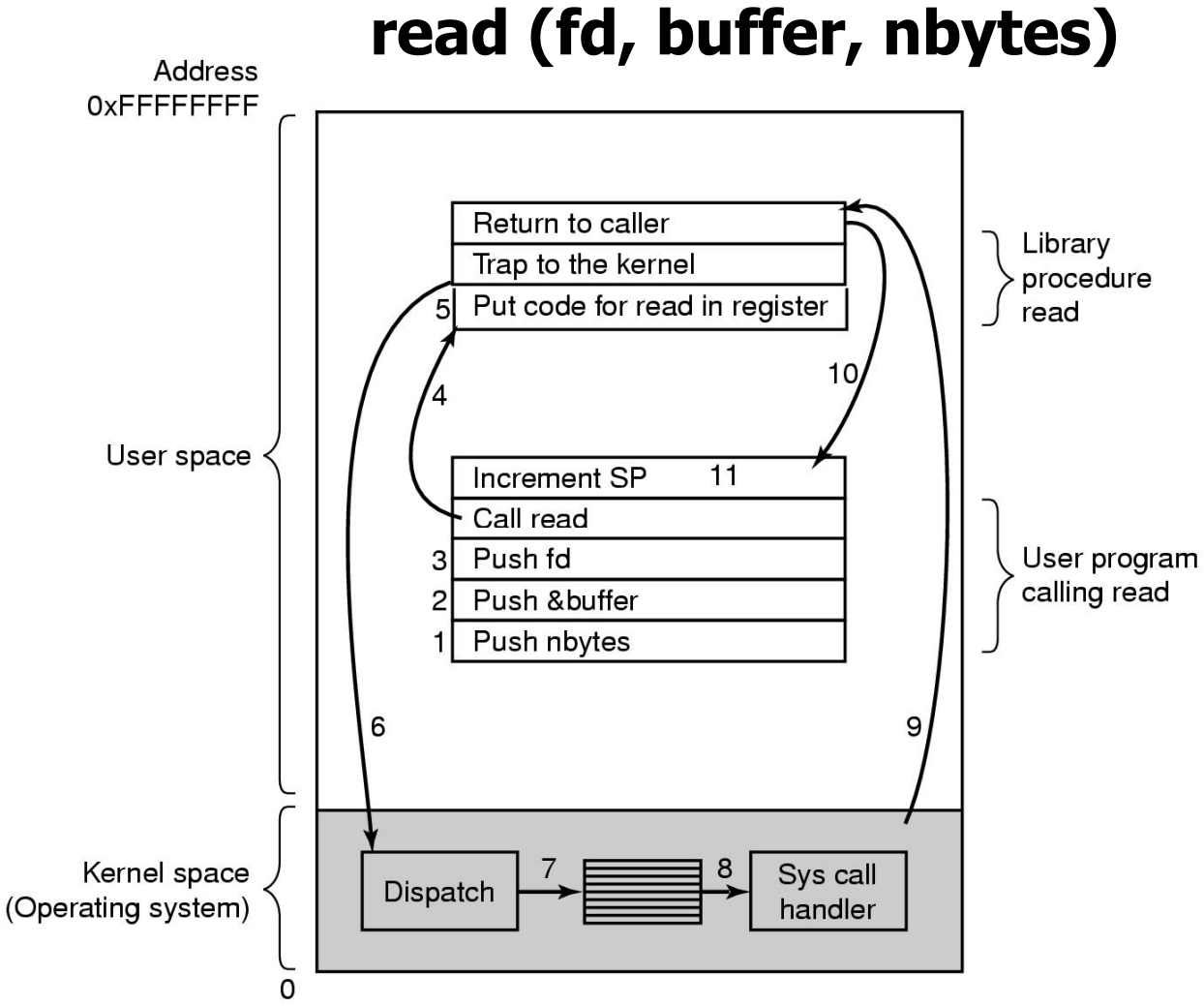
| Signal | Description | default action |
|---|---|---|
| **SIGSEGV** | **Invalid memory reference** | **implementation dependent** |
| SIGSTOP | Execution stopped | stop |
| **SIGTERM** | **termination** | **Abnormal termination** |
| SIGTSTP | Terminal stop | stop |
| SIGTTIN | Background process attempting read | stop |
| SIGTTOU | Background process attempting write | stop |
| **SIGURG** | **High bandwidth data available on socket** | **ignore** |
| **SIGUSR1** | **User-defined signal 1** | **abnormal termination** |

# Command Line Generates Signals



- You can send a signal to a process from the command line using **kill**
- kill -l will list the signals the system understands
- kill [-signal] pid will send a signal to a process.
  - The optional argument may be a name or a number (default is SIGTERM).
- To unconditionally kill a process, use:
  - kill -9 pid which is
    kill -SIGKILL pid.

# Steps in Making a System Call



# Some System Calls For Process Management

**Process management**

| Call | Description |
|---|---|
| pid = fork( ) | Create a child process identical to the parent |
| pid = waitpid(pid, &statloc, options) | Wait for a child to terminate |
| s = execve(name, argv, environp) | Replace a process' core image |
| exit(status) | Terminate process execution and return status |

# Some System Calls For File Management

**File management**

| Call | Description |
|---|---|
| fd = open(file, how, ...) | Open a file for reading, writing or both |
| s = close(fd) | Close an open file |
| n = read(fd, buffer, nbytes) | Read data from a file into a buffer |
| n = write(fd, buffer, nbytes) | Write data from a buffer into a file |
| position = lseek(fd, offset, whence) | Move the file pointer |
| s = stat(name, &buf) | Get a file's status information |

# Some System Calls For Directory Management

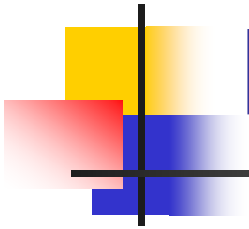
**Directory and file system management**

| Call | Description |
|---|---|
| s = mkdir(name, mode) | Create a new directory |
| s = rmdir(name) | Remove an empty directory |
| s = link(name1, name2) | Create a new entry, name2, pointing to name1 |
| s = unlink(name) | Remove a directory entry |
| s = mount(special, name, flag) | Mount a file system |
| s = umount(special) | Unmount a file system |

# Some System Calls For Miscellaneous Tasks

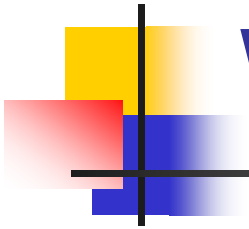
**Miscellaneous**

| Call | Description |
|---|---|
| s = chdir(dirname) | Change the working directory |
| s = chmod(name, mode) | Change a file's protection bits |
| s = kill(pid, signal) | Send a signal to a process |
| seconds = time(&seconds) | Get the elapsed time since Jan. 1, 1970 |



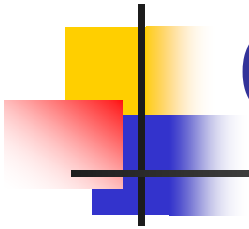
# Hardware Interrupts

- Hardware generated:
  - Different I/O devices are connected to different physical lines (pins) of an “Interrupt controller”
  - Device hardware signals the corresponding line
  - Interrupt controller signals the CPU (by signaling the Interrupt pin and passes an interrupt number
  - CPU saves return address after next instruction and jumps to corresponding interrupt handler



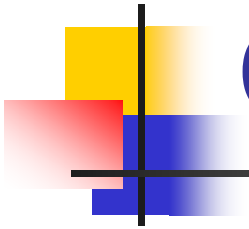
# Why Hardware Interrupts?

- Hardware devices may need asynchronous and immediate service. For example:
  - Timer interrupt: Timers and time-dependent activities need to be updated with the passage of time at precise intervals
  - Network interrupt: The network card interrupts the CPU when data arrives from the network
  - I/O device interrupt: I/O devices (such as mouse and keyboard) issue hardware interrupts when they have input (e.g., a new character or mouse click)



# Other Interrupts

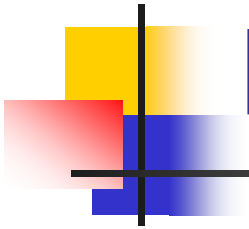
- Software Interrupts:
  - Interrupts caused by the execution of a software instruction:
    - INT <interrupt_number>
  - Example: The system call interrupt
- Initiated by the running (user level) process
- Cause current processing to be interrupted and transfers control to the corresponding interrupt handler in the kernel



# Other Interrupts

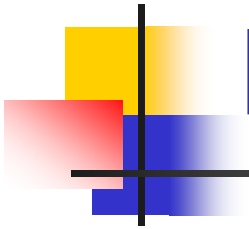
- Exceptions
  - Initiated by processor hardware itself
  - Example: divide by zero
- Like a software interrupt, they cause a transfer of control to the kernel to handle the exception

# Registering an Interrupt Handler (top half)

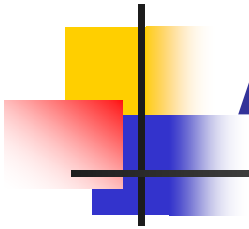


- request_irq (irq, handler, flags, name, dev)
- free_irq (irq, dev)
- Notes:
  - *handler* is a pointer to the interrupt handler function
  - Interrupt handlers need not be re-entrant (same irq is masked until handler exits)
  - IRQ lines can be shared by multiple devices. The parameter dev is a unique "cookie" to be supplied by the given device and checked by the handler
  - The kernel sequentially invokes all handlers registered for a given irq

# Interrupts, Priorities and Blocking

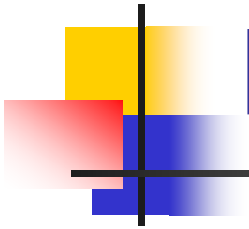


- Interrupts (as the name suggests) have the highest priority (compared to user and kernel threads) and therefore run first
  - What are the implications on regular program execution?
    - Must keep interrupt code short in order not to keep other processing stopped for a long time
    - Cannot block (regular processing does not resume until interrupt returns, so if the interrupt blocks in the middle the system "hangs")



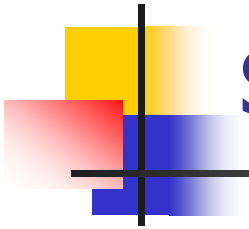
# A Note on Multicore

- How are interrupts handled on multicore machines?
  - On x86 systems each CPU gets its own local Advanced Programmable Interrupt Controller (APIC). They are wired in a way that allows routing device interrupts to any selected local APIC.
  - The OS can program the APICs to determine which interrupts get routed to which CPUs.
  - The default (unless OS states otherwise) is to route all interrupts to processor 0



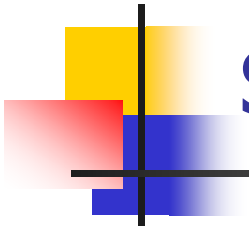
# Bottom Halves

- Since the interrupt handler must be minimal, all other processing related to the event that caused the interrupt must be deferred
  - Example:
    - Network interrupt causes packet to be copied from network card
    - Other processing on the packet should be deferred until its time comes
- The deferred portion of interrupt processing is called the "Bottom Half"



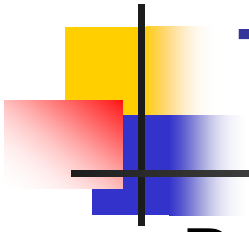
# soft_irq

- 32 handlers that must be statically defined in the Linux kernel.
- A hardware interrupt (before returning) uses raise_softirq() to mark that a given soft_irq must execute the bottom half
- At a later time, when scheduling permits, the marked soft_irq handler is executed
  - When a hardware interrupt is finished
  - When a process makes a system call
  - When a new process is scheduled



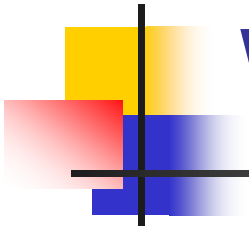
# soft_irq Types

- HI_SOFTIRQ
- TIMER_SOFTIRQ
- NET_TX_SOFTRQ
- NET_RX_SOFTIRQ
- BLOCK_SOFTIRQ
- TASKLET_SOFTIRQ
- SCHED_SOFTIRQ
- …



# Tasklets

- Bottom halves multiplexed on top of soft_irq's
- Scheduled using
  - tasklet_schedule()
  - tasklet_hi_schedule()
- Same tasklet invocations are serialized
- Tasklets can be created or removed dynamically
- Cannot sleep (cannot save their context)

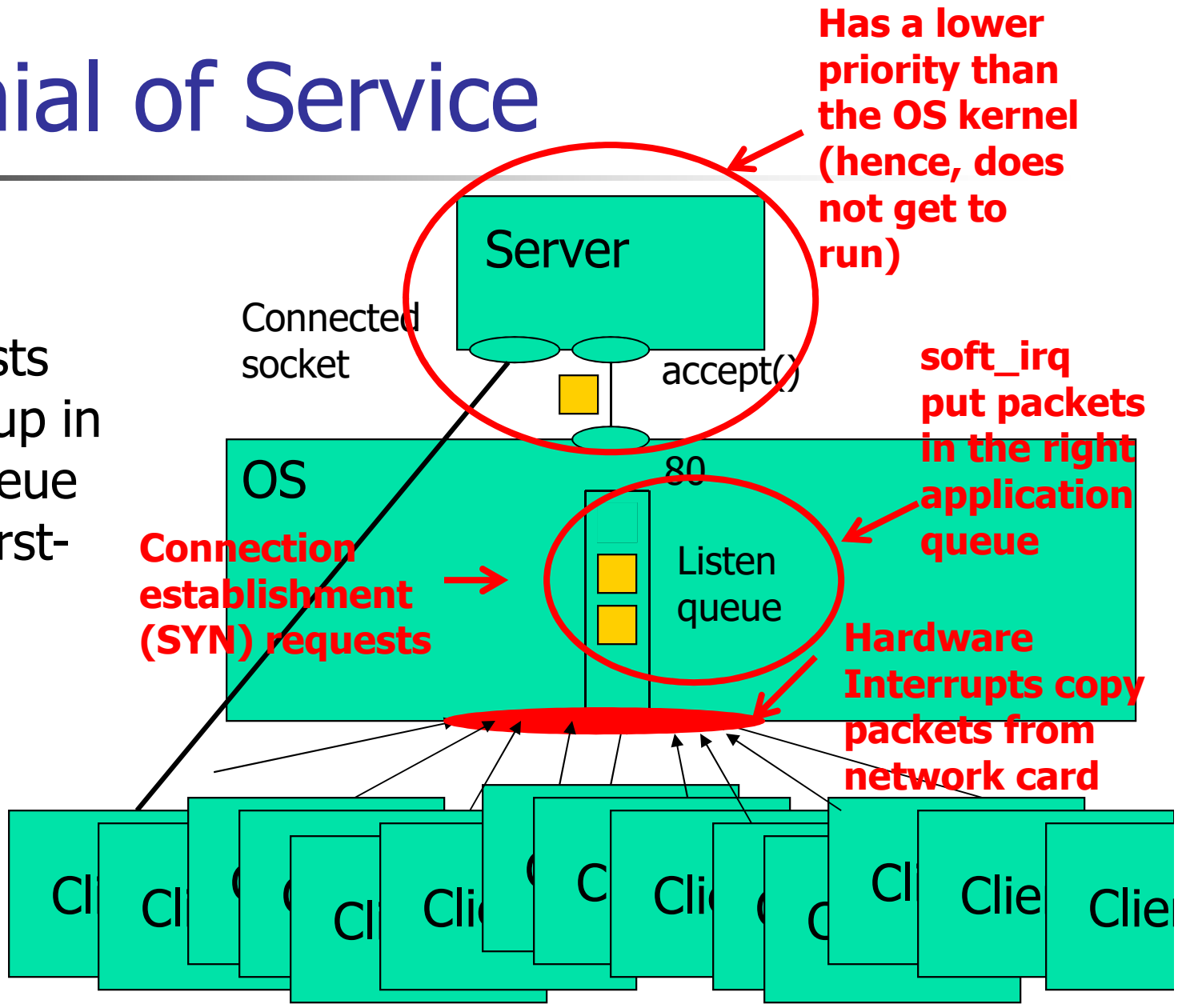


# Work Queues

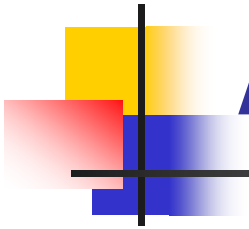
- Work deferred to its own thread
- Can be scheduled together with other threads according to priorities set by a scheduling policy
- Associated with its thread control block and hence can block (and save context)

# Denial of Service

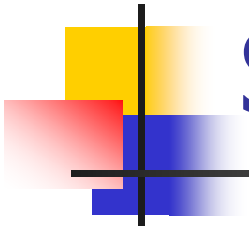
Client requests get queued-up in the listen queue
First-come first-served



# What Scheduling Policies are Available in Linux?



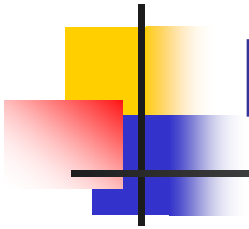
- SCHED_FIFO, SCHED_RR, SCHED_NORMAL, SCHED_FAIR (2.6+), …
- What is nice()?
- What are priorities?
- What are real-time priorities?



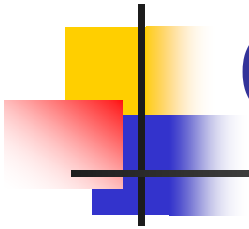
# Scheduler API

| System call | description |
|---|---|
| nice() | change the priority |
| getpriority() | get the maximum group priority |
| setpriority() | set the group priority |
| sched_getscheduler() | get the scheduling policy |
| sched_setscheduler() | set the scheduling policy and priority |
| sched_getparam() | get the priority |
| sched_setparam() | set the priority |
| sched_yield() | relinquish the processor voluntarily |
| sched_get_priority_min() | get the minimum priority value |
| sched_get_priority_max() | get the maximum priority value |
| sched_rr_get_interval() | get the time quantum for Round-Robin |

# Process Priority & Timeslice Recalculation

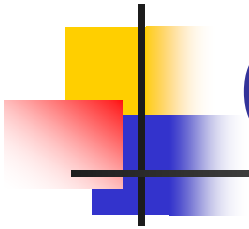


- Processes start at 120 by default
- Static priority
  - A "*nice*" value: 19 to -20.
  - Inherited from the parent process
  - Altered by user (negative values require special permission)
- Dynamic priority
  - Based on static priority and applications characteristics (interactive or CPU-bound)
  - Favor interactive applications over CPU-bound ones
- Timeslice is mapped from priority



# Completely Fair Scheduler

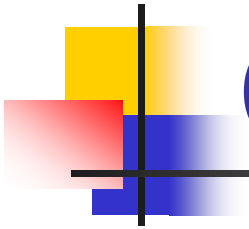
- Merged into the 2.6.23 release of the Linux kernel and is the default scheduler.
- Scheduler maintains a red-black tree where nodes are ordered according to received virtual execution time
- Node with smallest virtual received execution time is picked next
- Priorities determine accumulation rate of virtual execution time
  - Higher priority → slower accumulation rate



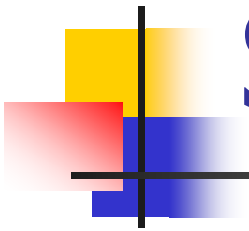
# Other Scheduling Policies (… that you can implement)

- What if you want to maximize throughput?
  - Shortest job first!

# Other Scheduling Policies (… that you can implement)



- What if you want to meet all deadlines?
  - Earliest deadline first!
- Problem?
  - Works only if you are not "overloaded". If the total amount of work is more than capacity, a domino effect occurs as you always choose the task with the nearest deadline (that you have the least chance of finishing by the deadline), so you may miss a lot of deadlines!



# Synchronization

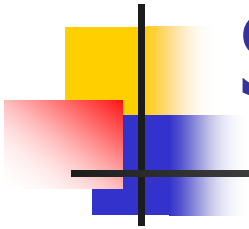
- Processes and threads can be preempted at arbitrary times, which may generate problems.
- Example: What is the execution outcome of the following two threads (initially x=0)?

**Thread 1:**

**Read X**
**Add 1**
**Write X**

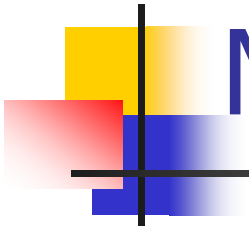
**Thread 2:**

**Read X**
**Add 1**
**Write X**



# Semaphores in POSIX

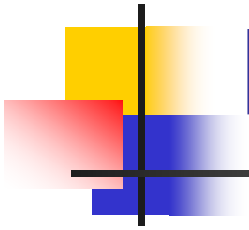
- int sem_init(sem_t *sem, int pshared, unsigned value);
- int sem_destroy(sem_t *sem);
- int sem_trywait(sem_t *sem);
- int sem_wait(sem_t *sem);
- int sem_post(sem_t *sem);



# Mutex (Lock)

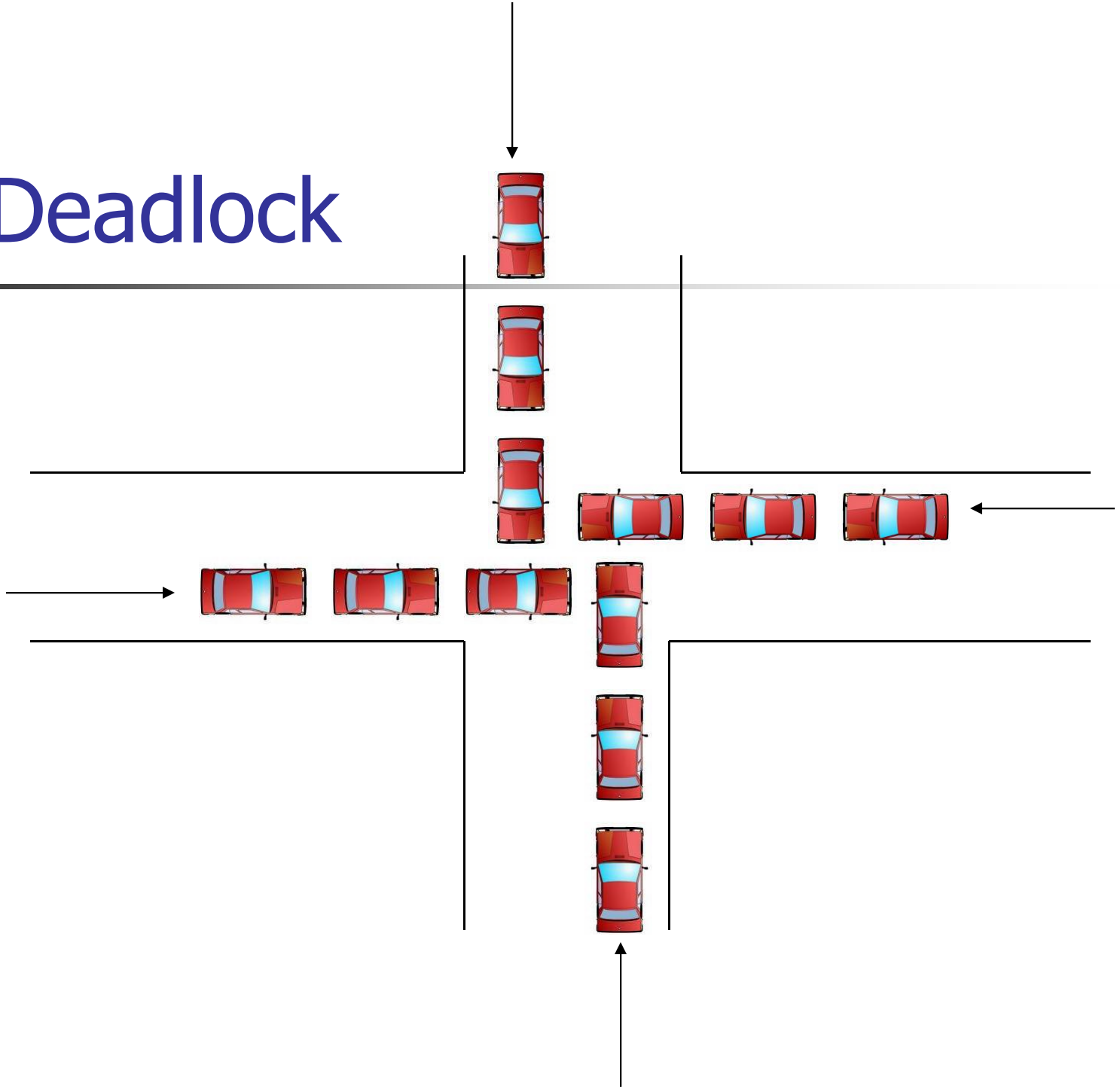
- Simplest and most efficient thread synchronization mechanism
- A special variable that can be either in
  - locked state: some thread *holds* or *owns* the *mutex;* or
  - unlocked state: no thread holds the *mutex*
- When several threads compete for a *mutex,* the losers block at that call
  - The *mutex* also has a queue of threads that are waiting to hold the *mutex.*

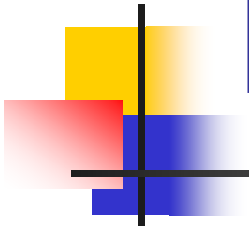
# POSIX Mutex-related Functions



- int **pthread_mutex_init**(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
  - Also see PTHREAD_MUTEX_INITIALIZER
- int **pthread_mutex_destroy**(pthread_mutex_t *mutex);
- int **pthread_mutex_lock**(pthread_mutex_t *mutex);
- int **pthread_mutex_trylock**(pthread_mutex_t *mutex);
- int **pthread_mutex_unlock**(pthread_mutex_t *mutex);

# Deadlock

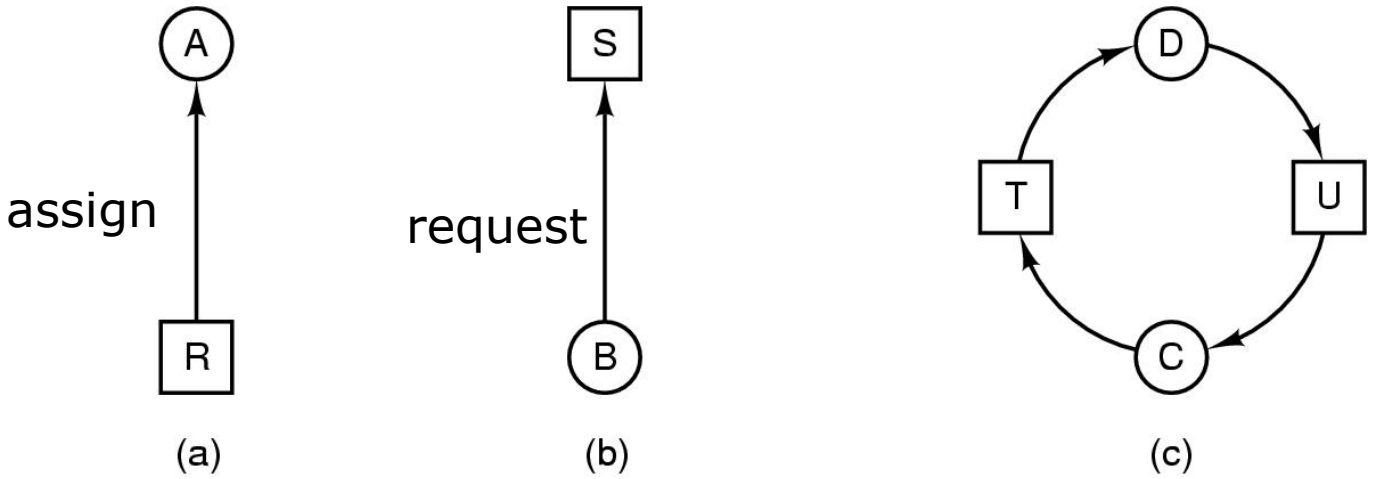




# Deadlock

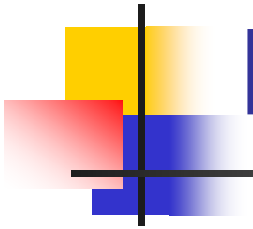
- Mutual exclusion
- Hold and wait condition
- No preemption condition
- Circular wait condition

# Resource Allocation Graph

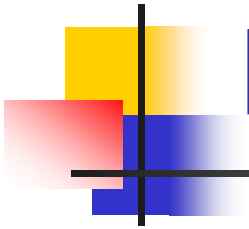



- resource R assigned to process A
- process B is requesting/waiting for resource S
- process C and D are in deadlock over resources T and U

# Real-time Scheduling of Periodic Tasks



- Result #1: Earliest Deadline First (EDF) is the optimal dynamic priority scheduling policy for independent periodic tasks (meets the most deadlines of all dynamic priority scheduling policies)
- Result #2: Rate Monotonic Scheduling (RM) is the optimal static priority scheduling policy for independent periodic tasks (meets the most deadlines of all static priority scheduling policies)

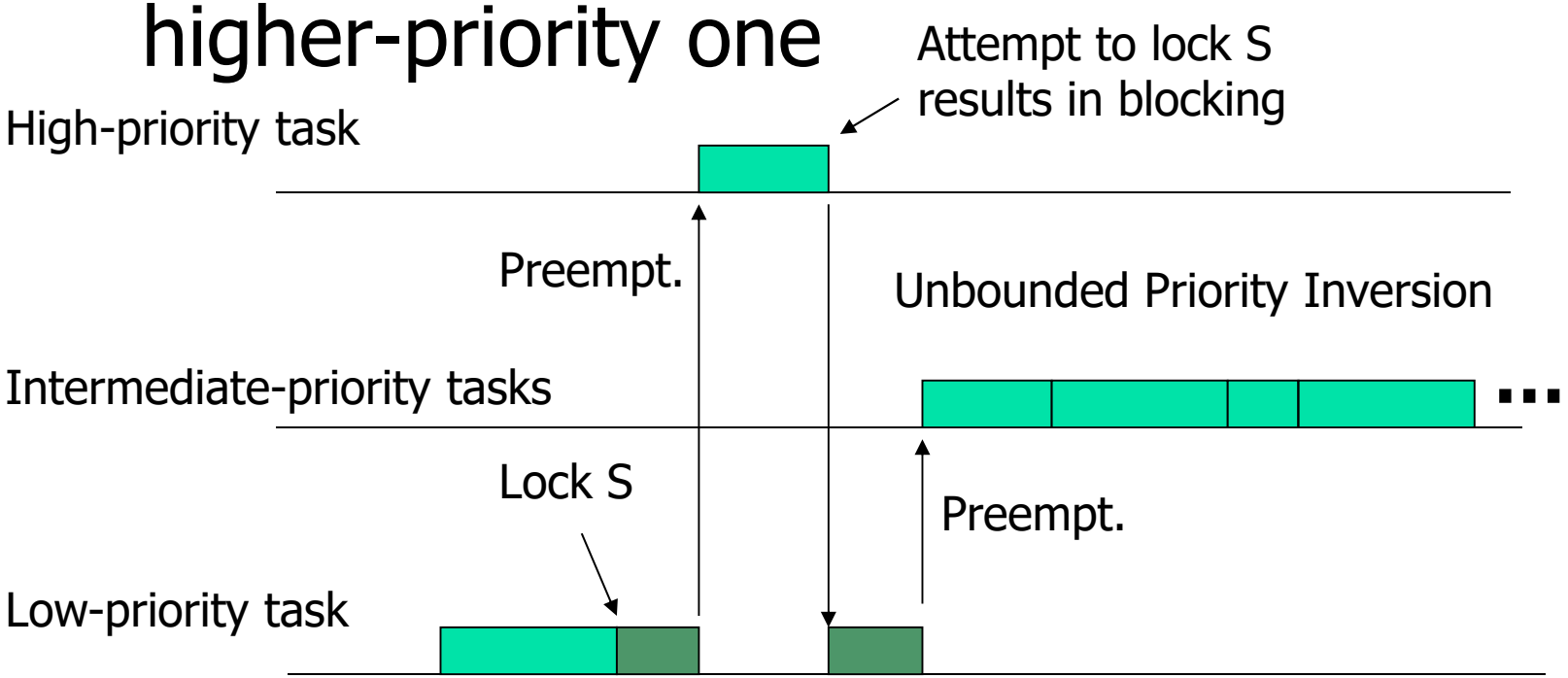


# Advanced Topic: Locking and Priority Inversion

- What if a higher-priority process needs a resource locked by a lower-priority process?
  - How long will the higher priority process have to wait for lower-priority execution?

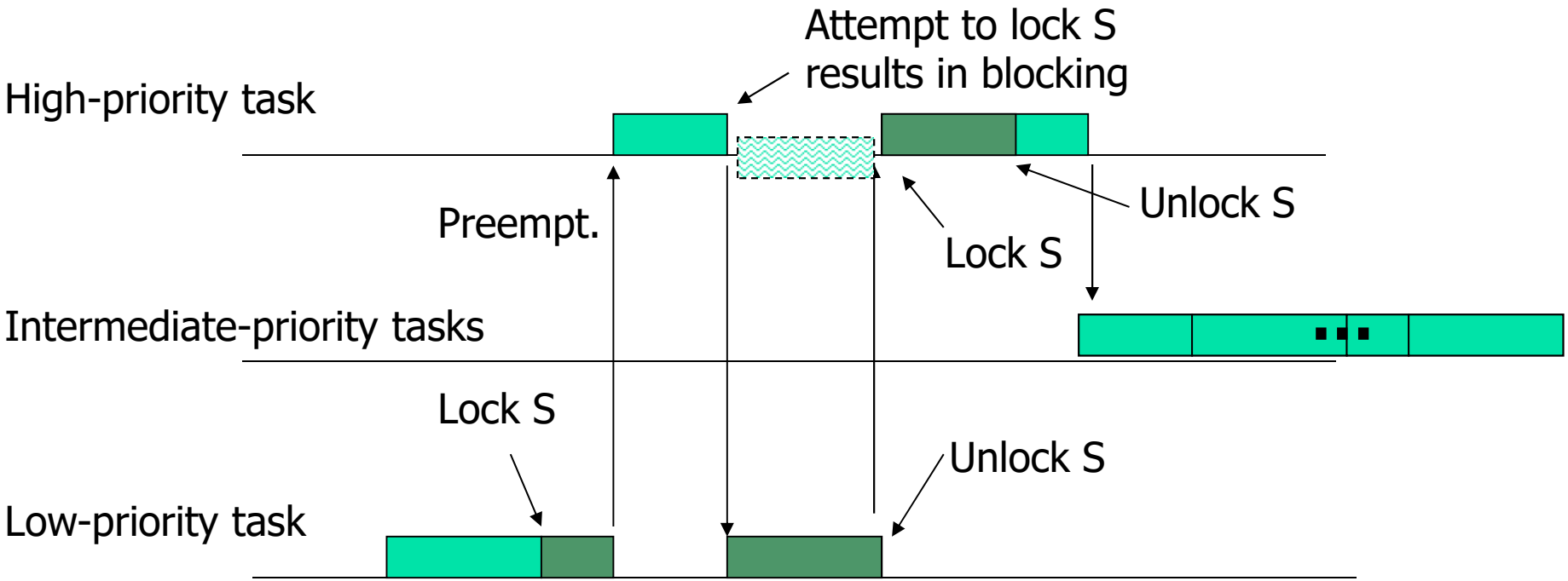
# Unbounded Priority Inversion

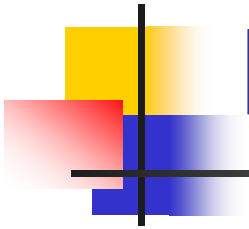
- Consider the case below: a series of intermediate priority tasks is delaying a higher-priority one



# Priority Inheritance Protocol

- Let a task inherit the priority of any higher-priority task it is blocking



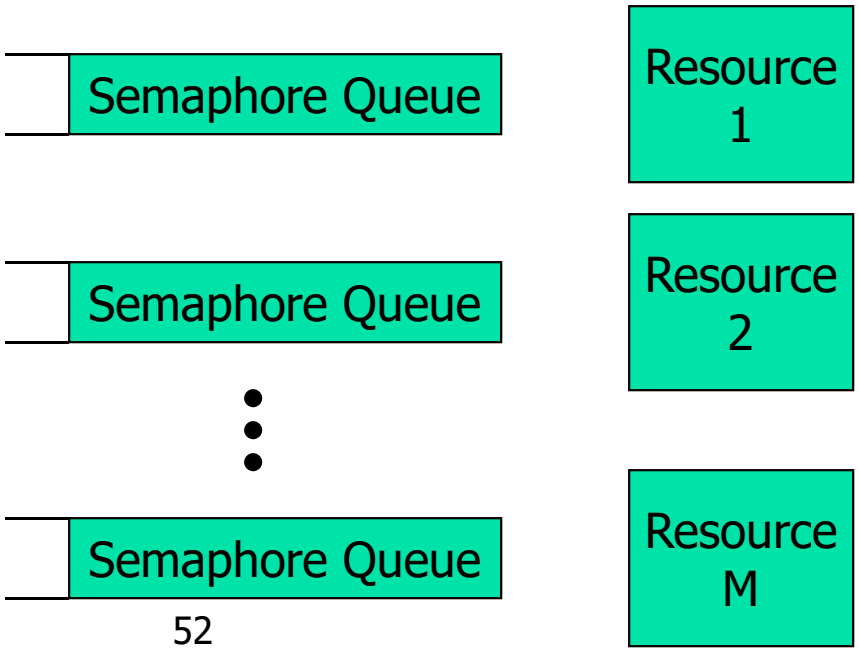


# Priority Inheritance Protocol

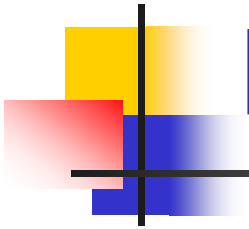
- Question: What is the longest time a task can wait for lower-priority tasks?
  - Let there be $N$ tasks and $M$ locks
  - Let the largest critical section of task $i$ be of length $B_i$
- Answer: ?

# Computing the Maximum Priority Inversion Time

- Consider the instant when a high-priority task that arrives.
  - What is the most it can wait for lower priority ones?



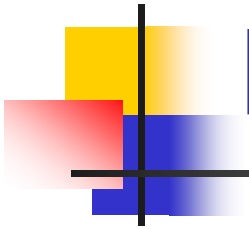

If I am a task, priority inversion occurs when
(a) Lower priority task holds a resource I need (direct blocking)
(b) Lower priority task inherits a higher priority than me because it holds a resource the higher-priority task needs (push-through blocking)



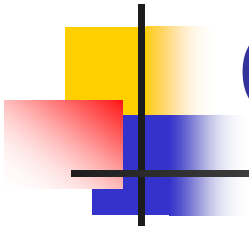
# Priority Ceiling Protocol

- Definition: The priority ceiling of a semaphore is the highest priority of any task that can lock it
- A task that requests a lock $R_k$ is denied if its priority is not higher than the highest priority ceiling of all currently locked semaphores (say it belongs to semaphore $R_h$)
  - The task is said to be blocked by the task holding lock $R_h$
- A task inherits the priority of the top higher-priority task it is blocking

# Advanced Configuration and Power Interface (ACPI)



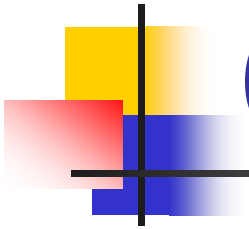
- Defines different power saving states in a platform-independent manner
- The standard was originally developed by Intel, Microsoft, and Toshiba (in 1996), then later joined by HP, and Phoenix.
- The latest version is "Revision 5.0," published in 2011.



# Global States

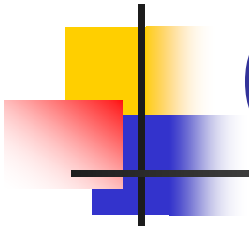
- **G0**: *working*
- **G1**: *Sleeping* and *hibernation* (several degrees available)
- **G2**:, *Soft Off*: almost the same as G3 *Mechanical Off*, except that the power supply still supplies power, at a minimum, to the power button to allow wakeup. A full reboot is required.
- **G3**, *Mechanical Off*: The computer's power has been totally removed via a mechanical switch (as on the rear of a PSU).

# Processor "Sleep" States (C-states)

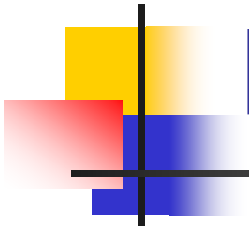


- **C0**: is the operating state.
- **C1** (often known as *Halt*): is a state where the processor is not executing instructions, but can return to an executing state instantaneously. All ACPI-conformant processors must support this power state.
- **C2** (often known as *Stop-Clock*): is a state where the processor maintains all software-visible state, but may take longer to wake up. This processor state is optional.
- **C3** (often known as *Sleep*) is a state where the processor does not need to keep its cache, but maintains other state. This processor state is optional.

# Processor Performance States (P-States)



- **P0** max power and frequency
- **P1** less than P0, voltage/frequency scaled
- **P2** less than P1, voltage/frequency scaled
- …
- **Pn** less than *P(n-1)*, voltage/frequency scaled



# Power of Computation

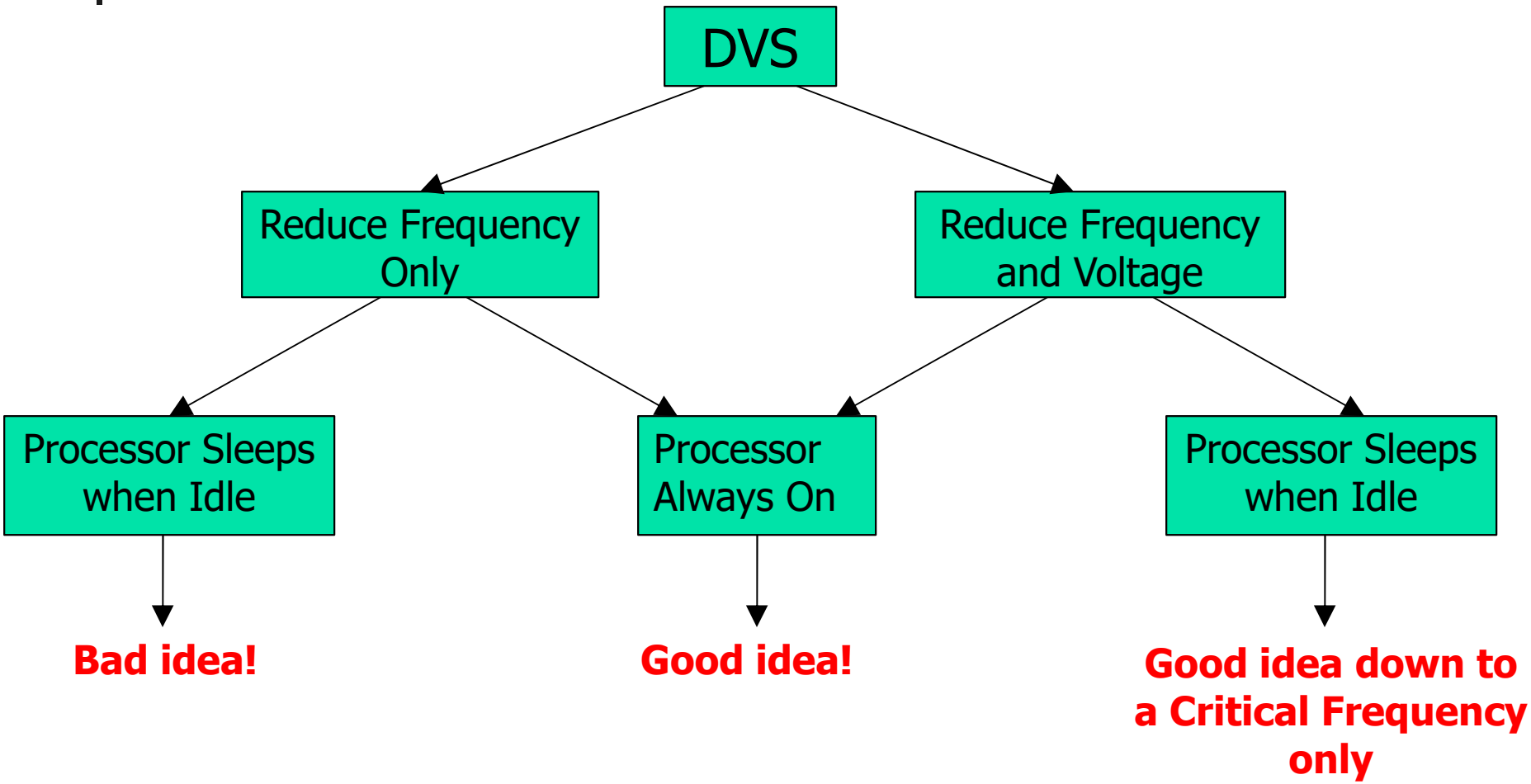
- Terminology
  - $R$ : Power spent on computation
  - $V$ : Processor voltage
  - $f$ : Processor clock frequency
  - $R_0$ : Leakage power
- Power spent on computation is:
  - $R = k_v V^2 f + R_0$

    where $k_v$ is a constant

# Energy of Computation

- Power spent on computation is:
  - $R = k_v V^2 f + R_0$
- Consider a piece of computation of length $C$ clock cycles and a processor operating at frequency $f$
- The execution time is $t = C/f$
- Energy spent is:
  - $E = R\,t = (k_v V^2 f + R_0)(C/f)$

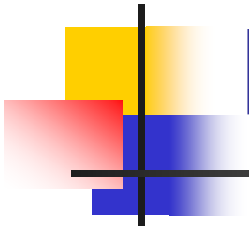
# Recap


DVS
Reduce Frequency Only
Reduce Frequency and Voltage
Processor Sleeps when Idle
Processor Always On
Processor Sleeps when Idle
Bad idea!
Good idea!
Good idea down to a Critical Frequency only

# Practical Consideration: Accounting for Off-chip Overhead

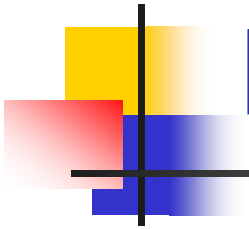
- In the preceding discussion, we assumed that task execution *time* at frequency $f$ is $C/f$, where $C$ is the total cycles needed
- In reality some cycles are lost waiting for memory access and I/O (Off-chip cycles).
  - Let the number of CPU cycles used be $C_{cpu}$ and the time spent off-chip be $C_{off\text{-}chip}$
  - Execution time at frequency $f$ is given by

$$C_{cpu}/f + C_{off\text{-}chip}$$



# Linux CPUFreq Governor

- Linux defines multiple DVS modes (called CPUfreq "governors"):
  - Performance (highest frequency)
  - Powersave (lowest frequency)
  - Userspace ("root" user controls frequency)
  - OnDemand (adaptively change frequency depending on load)



# Dynamic Power Management

- DPM refers to turning devices off (or putting them in deep sleep modes)
- Device wakeup has a cost that imposes a minimum sleep interval (a breakeven time)
- DPM must maximize power savings due to sleep while maintaining schedulability

# Turning Processors Off
## The Cost of Wakeup

- Energy expended on wakeup, $E_{wake}$
- To sleep or not to sleep?
  - Not to sleep (for time $t$):

$$E_{no\text{-}sleep} = (k_v V^2 f + R_0)\, t$$

  - To sleep (for time $t$) then wake up:

$$E_{sleep} = P_{sleep}\, t + E_{wake}$$

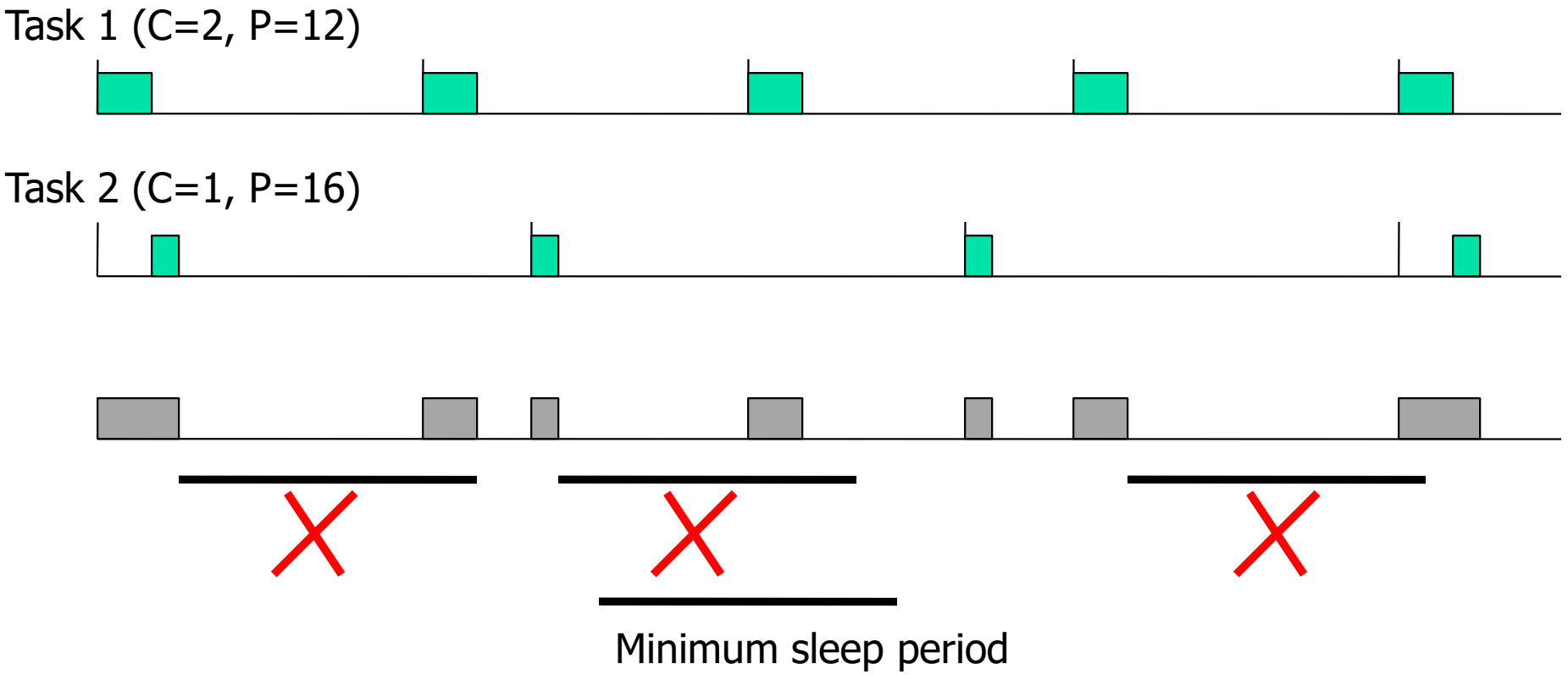
  - To save energy by sleeping: $E_{sleep} < E_{no\text{-}sleep}$

$$t > \frac{E_{wake}}{k_v V^2 f + R_0 - P_{sleep}}$$

**Minimum sleep interval**

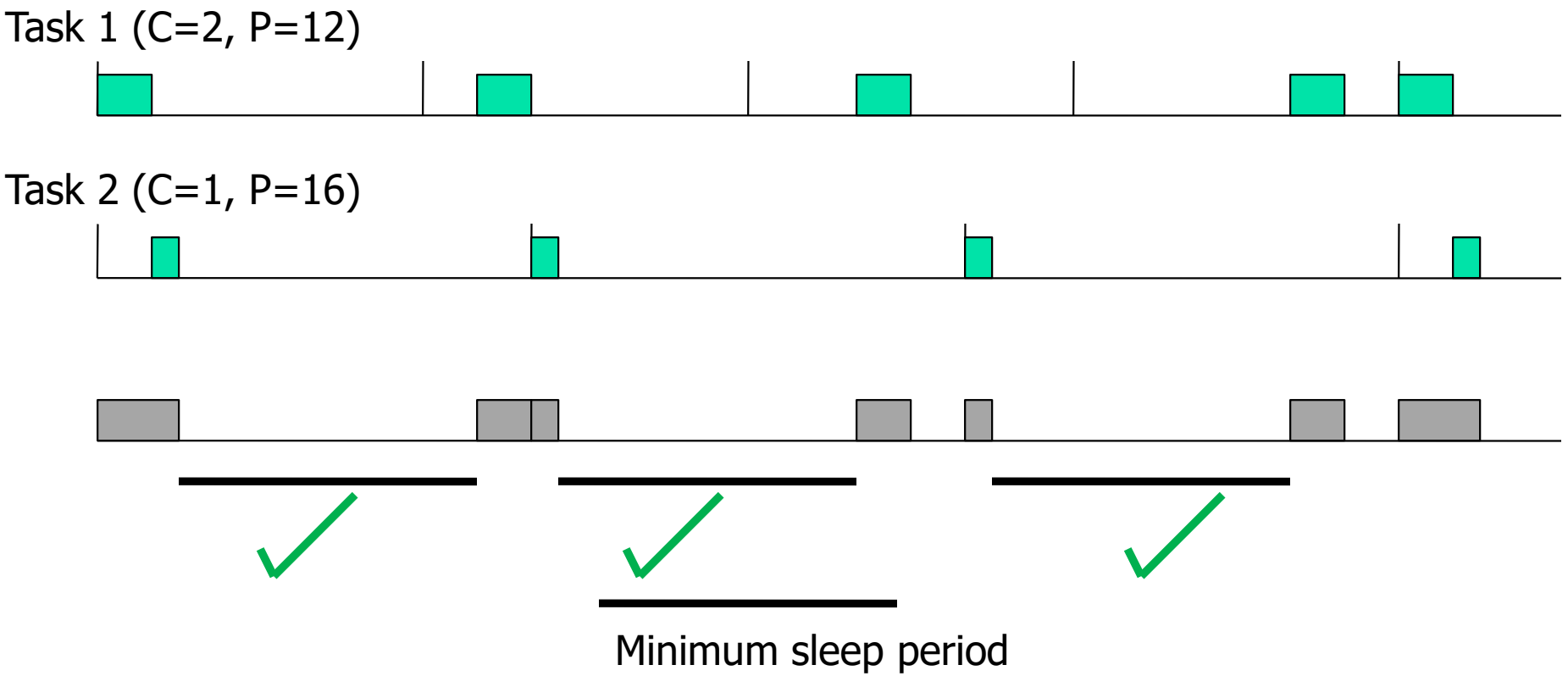
# DPM and the Problem with Work-conserving Scheduling

- No opportunity to sleep ☹

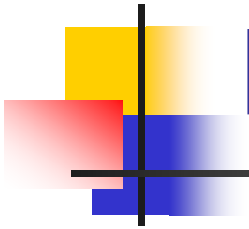


# DPM and the Problem with Work-conserving Scheduling

- Must batch! ☺



# How Many Processors to Use?



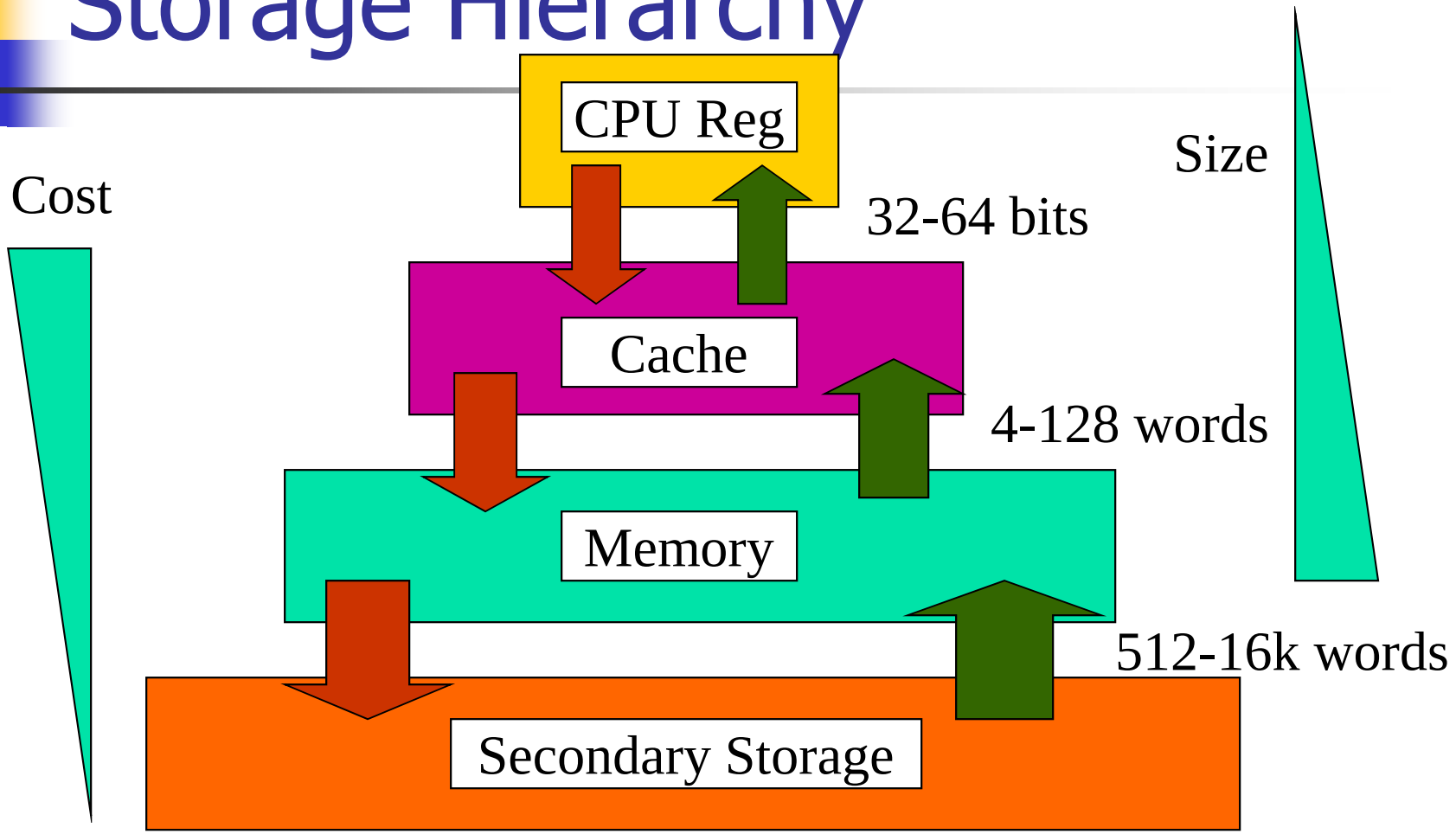
- Consider using one processor at frequency $f$ versus two at frequency $f/2$
- Case 1: Total power for one processor
  - $k_f f^3 + R_0$
- Case 2: Total power for two processors
  - $2\{k_f (f/2)^3 + R_0\} = k_f f^3/4 + 2R_0$
- The general case: $n$ processors
  - $n\{k_f (f/n)^3 + R_0\} = k_f f^3/n^2 + nR_0$

# How Many Processors to Use?

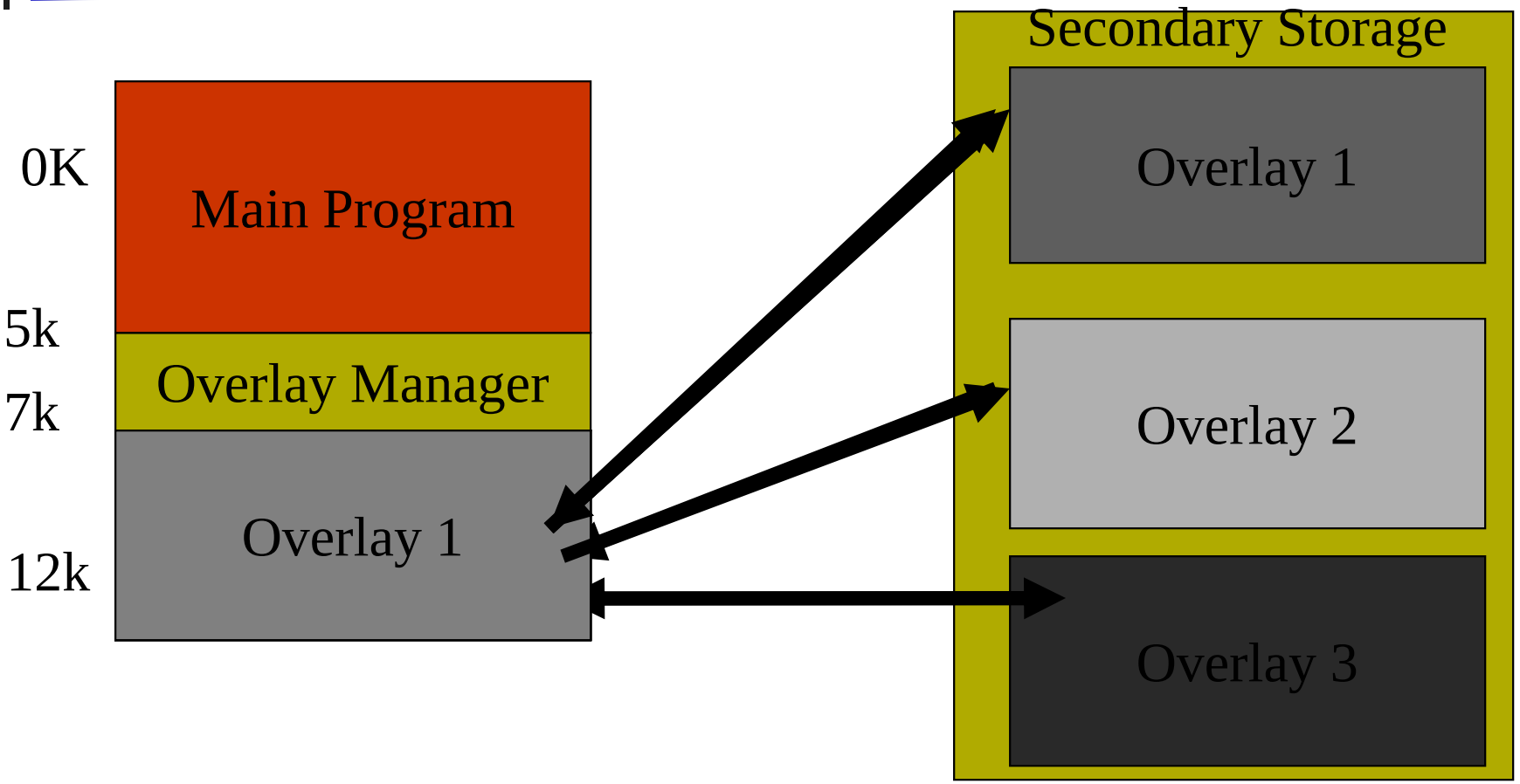
- The general case: $n$ processors
  - $Power = n \{k_f (f/n)^3 + R_0\} = k_f f^3/n^2 + n R_0$
  - $dPower/dn = -2 k_f f^3/n^3 + R_0 = 0$

$$n = \sqrt[3]{\frac{2k_f f^3}{R_0}}$$

# Storage Hierarchy



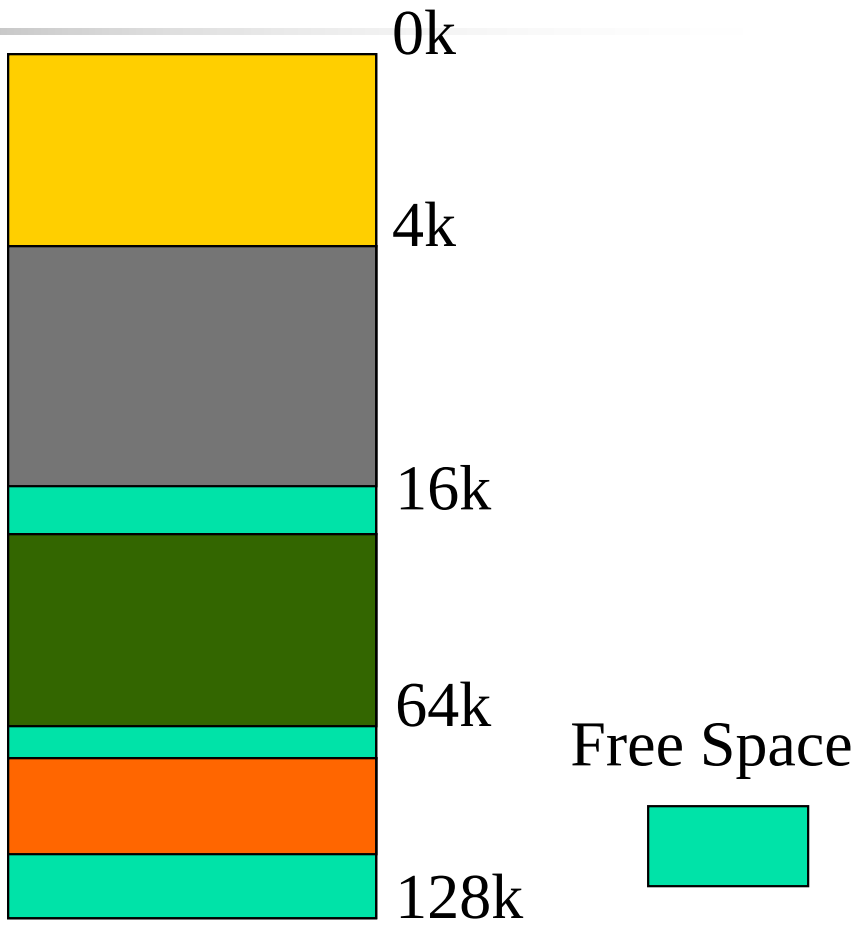
# History: Overlays



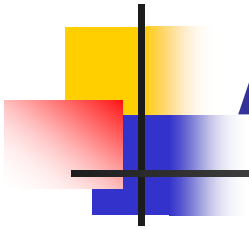

Used when process memory requirement exceeds the physical memory space

# History: Multiprogramming with Fixed Partitions

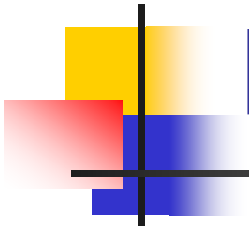
- Divide memory into *n* fixed (possibly unequal) partitions
- Problem:
  - Fragmentation



# History: Fixed Partition Allocation



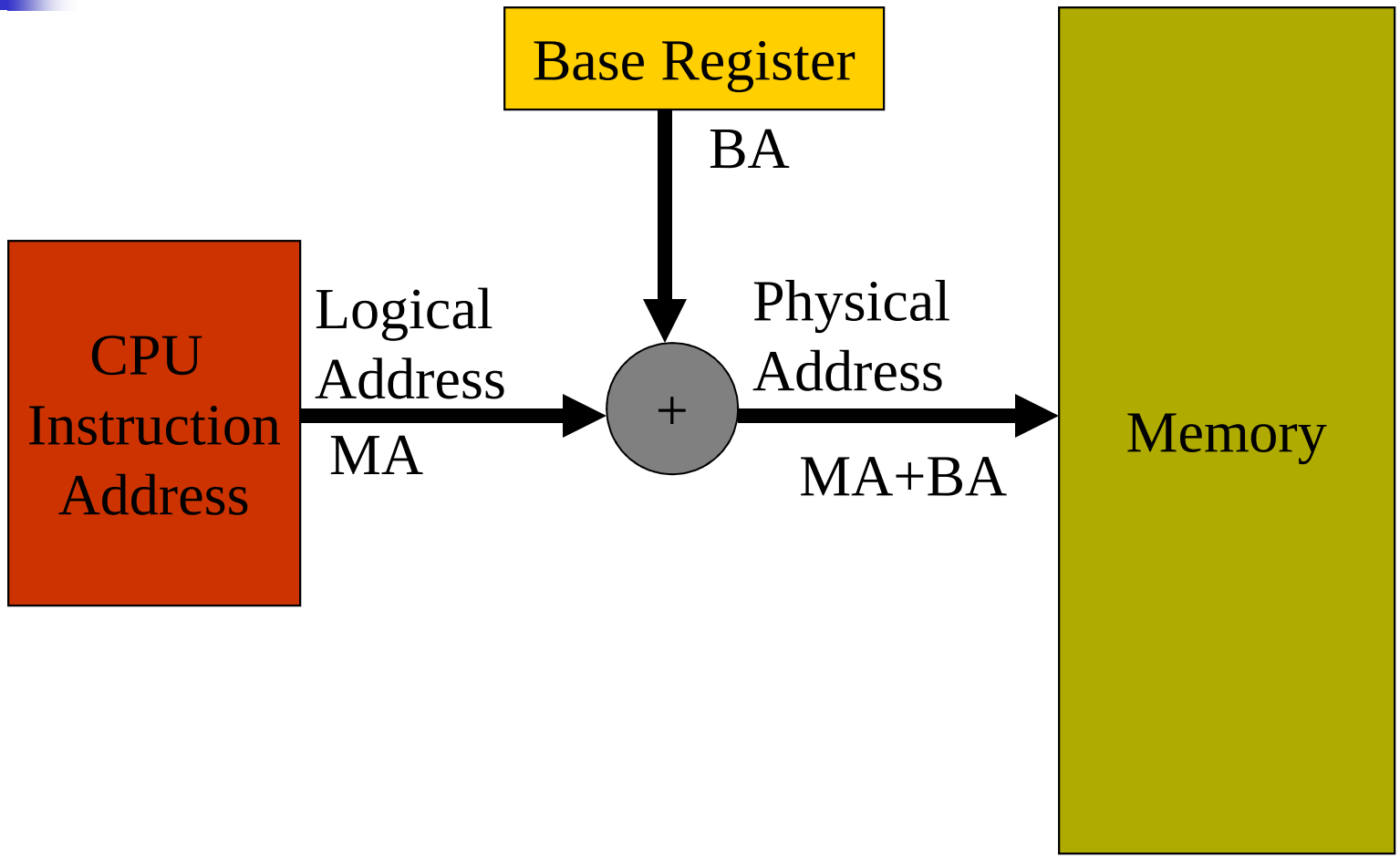
- Separate input queue for each partition
  - Sorting incoming jobs into separate queues
  - Inefficient utilization of memory
    - when the queue for a large partition is empty but the queue for a small partition is full. Small jobs have to wait to get into memory even though plenty of memory is free.
- One single input queue for all partitions.
  - Allocate a partition where the job fits in.
    - Best Fit, Worst Fit, First Fit



# History: Relocation

- Correct starting address when a program should start in the memory
- Different jobs will run at different addresses
  - When a program is linked, the linker must know at what address the program will begin in memory.
- Logical addresses
  - Logical address space , range (0 to max)
  - Physical addresses, Physical address space range (R+0 to R+max) for base value R.
  - User program never sees the real physical addresses
- Relocation register
  - Mapping requires hardware with the base register

# History: Relocation Register

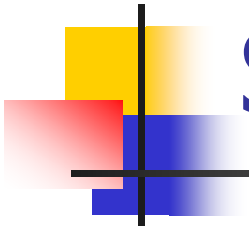


# History: Variable Partition Allocation

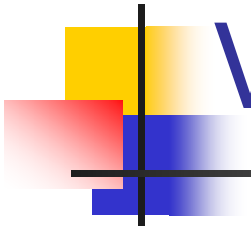



**Memory wasted by External Fragmentation**

# History: Storage Placement Strategies



- **Best Fit**
  - Use the hole whose size is equal to the need, or if none is equal, the hole that is larger but closest in size.
  - Problem: Creates small holes that can't be used.
- **First Fit**
  - Use the first available hole whose size is sufficient to meet the need.
  - Problem: Creates average size holes.
- **Next Fit**.
  - Minor variation of first fit: search from the last hole used.
  - Problem: slightly worse performance than first fit.
- **Worst Fit**.
  - Use the largest available hole.
  - Problem: Gets rid of large holes making it difficult to run large programs.

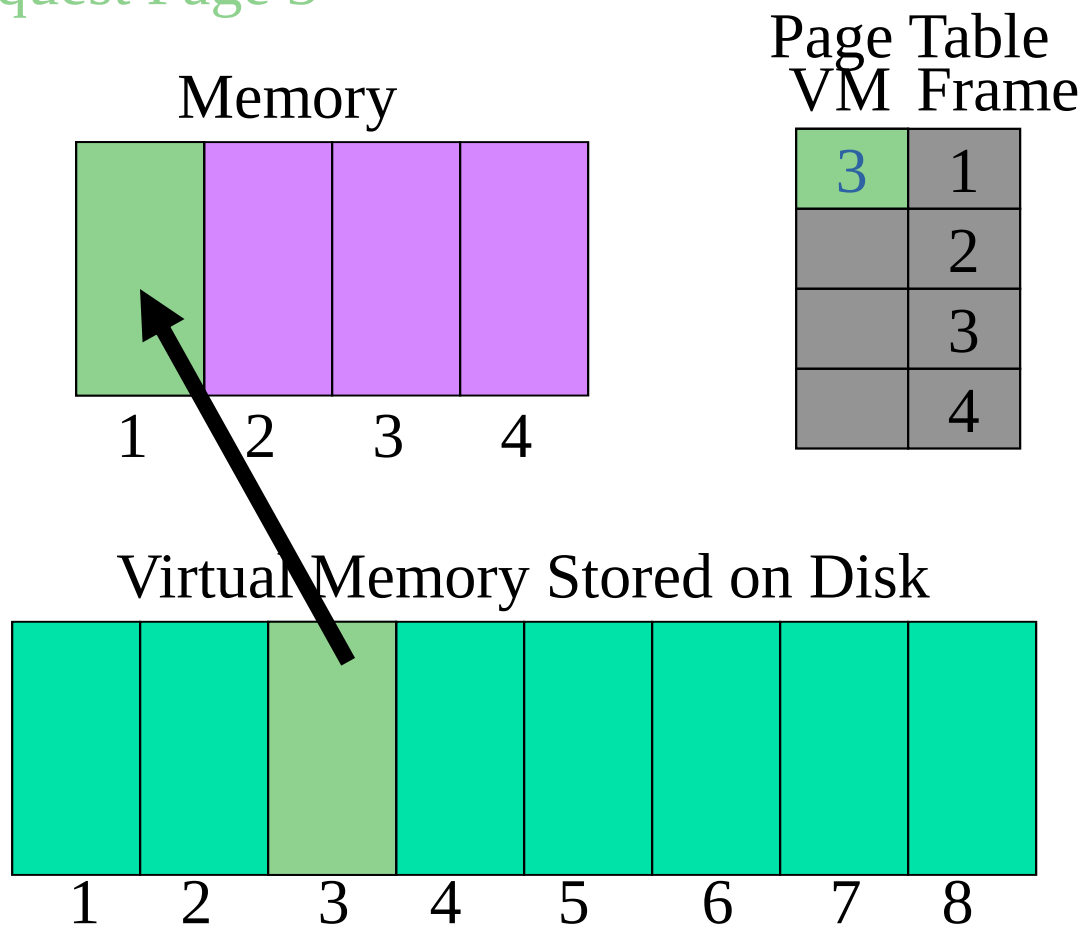


# Virtual Memory

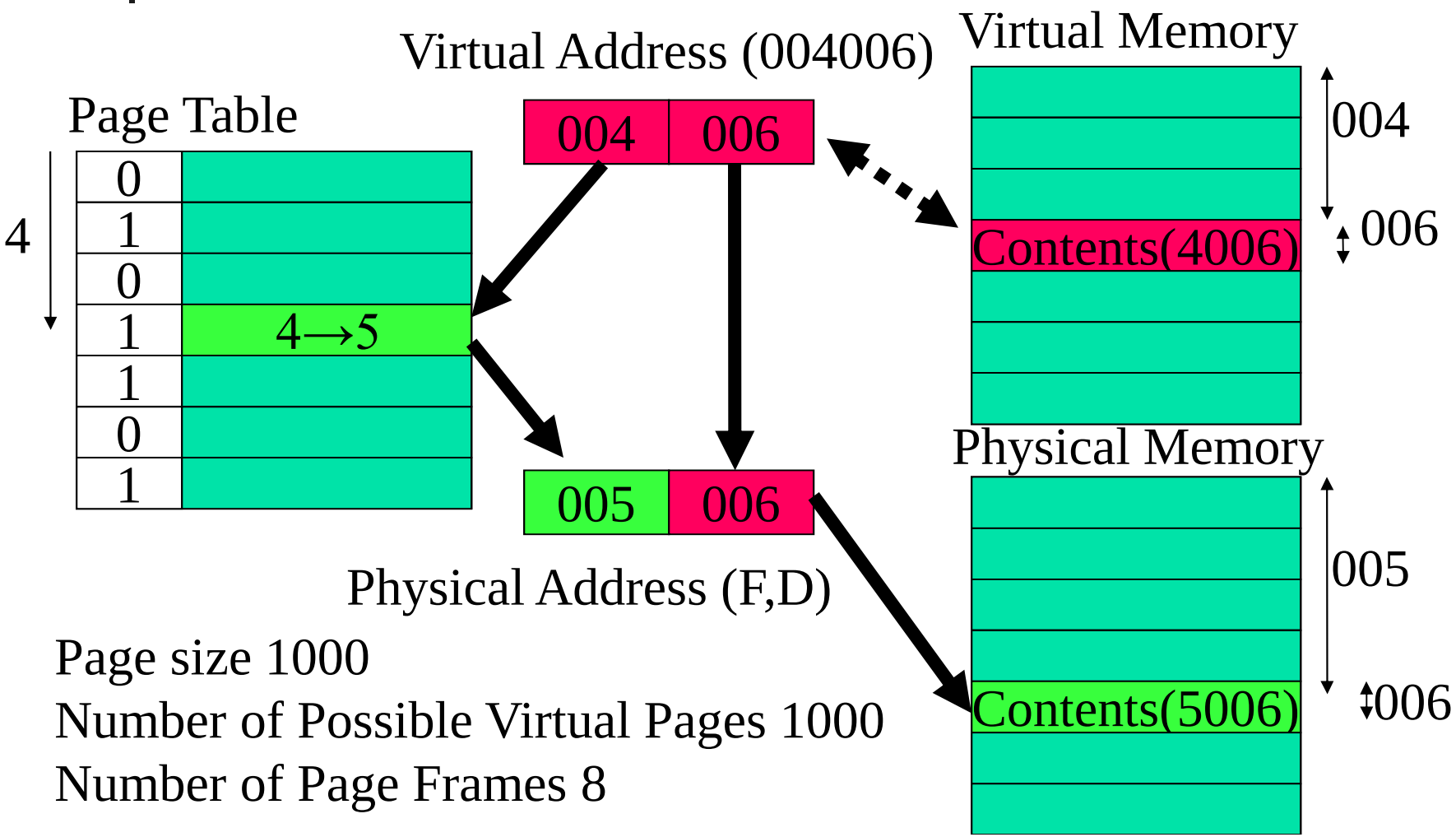
- Provide user with virtual memory that is as big as user needs
- Store virtual memory on disk
- Cache parts of virtual memory being used in real memory
- Load and store cached virtual memory without user program intervention

# Paging

Request Page 3



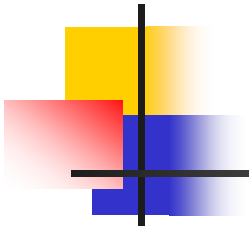
# Page Mapping Hardware




Page size 1000

Number of Possible Virtual Pages 1000

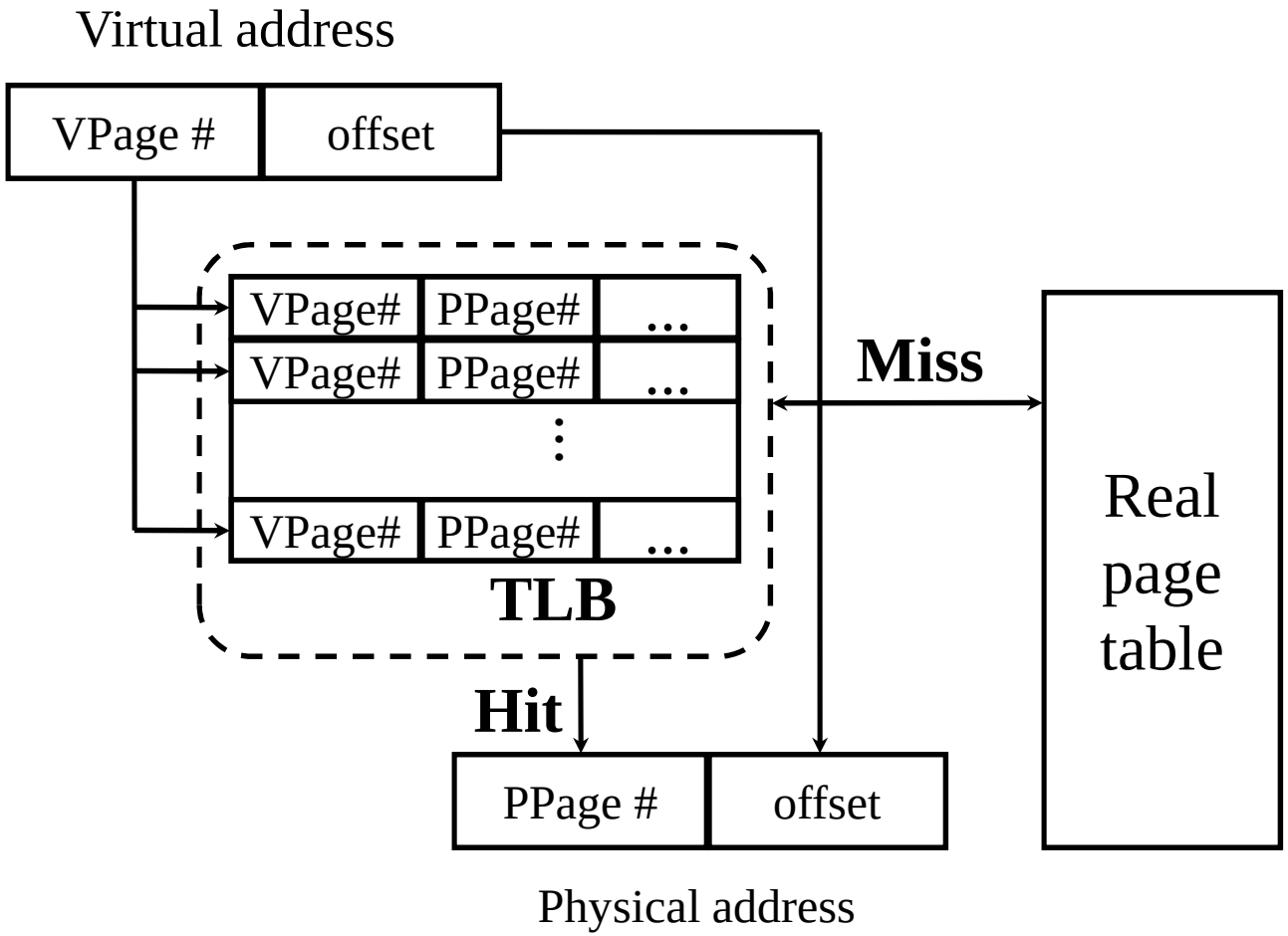
Number of Page Frames 8

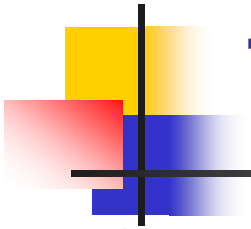


# Page Fault

- Access a virtual page that is not mapped into any physical page
  - A fault is triggered by hardware
- Page fault handler (in OS's VM subsystem)
  - Find if there is any free physical page available
    - If no, evict some resident page to disk (swapping space)
  - Allocate a free physical page
  - Load the faulted virtual page to the prepared physical page
  - Modify the page table

# Optimization: Translation Lookaside Buffer (TLB)

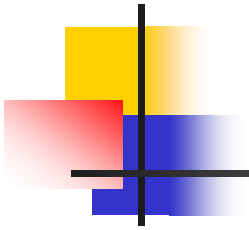




# TLB Function

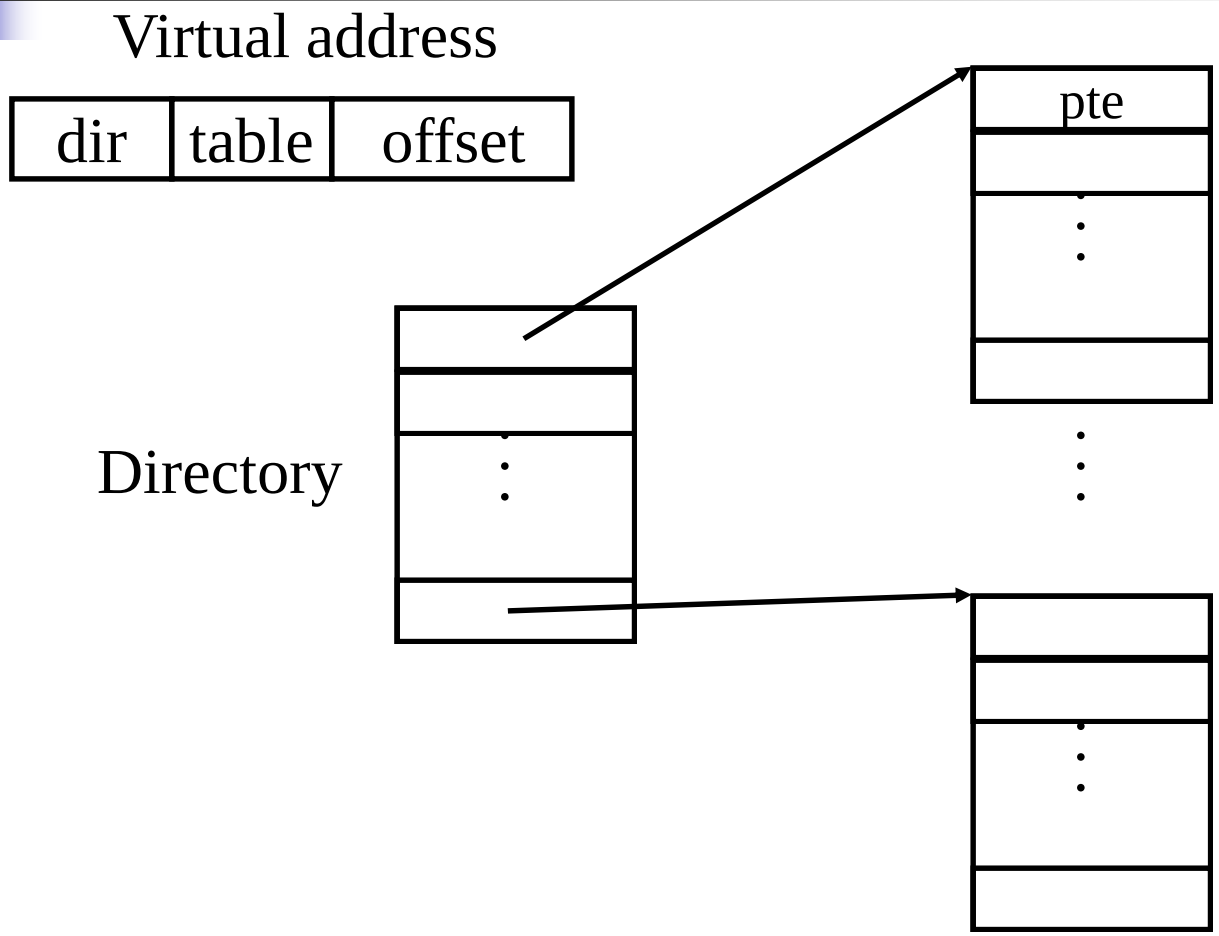
- If a virtual address is presented to MMU, the hardware checks TLB by comparing all entries simultaneously (in parallel).
- If match is valid, the page is taken from TLB without going through page table.
- If match is not valid
  - MMU detects miss and does a page table lookup.
  - It then evicts one page out of TLB and replaces it with the new entry, so that next time that page is found in TLB.

# Effective Access Time



- TLB lookup time = $\sigma$ time unit
- Memory cycle = $m$ µs
- TLB Hit ratio = $\eta$
- Effective access time
  - $Eat = (m + \sigma)\,\eta + (2m + \sigma)(1 - \eta)$
  - $Eat = 2m + \sigma - m\,\eta$

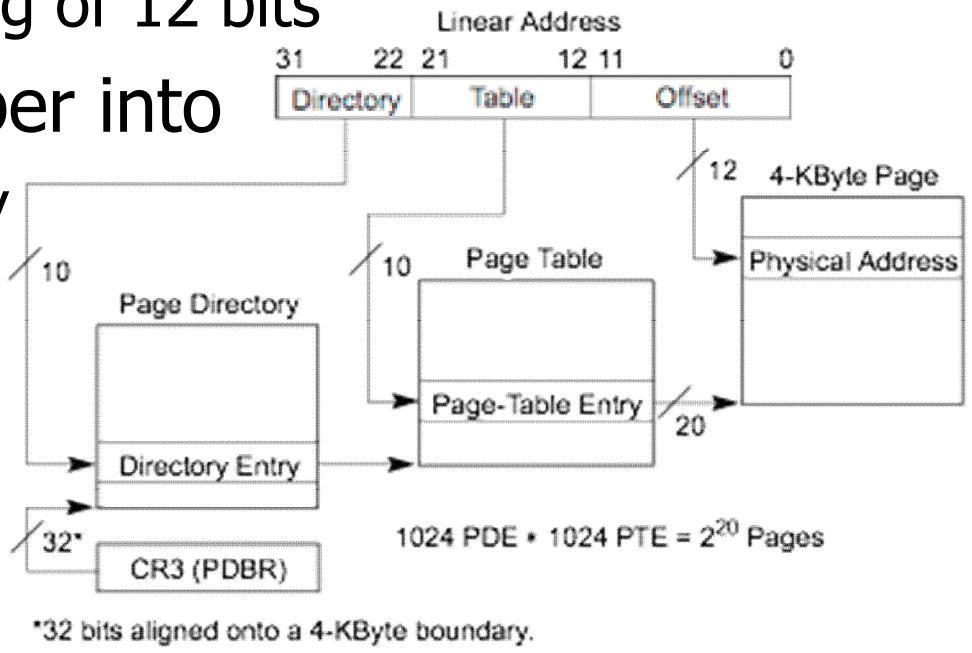
# Multilevel Page Tables



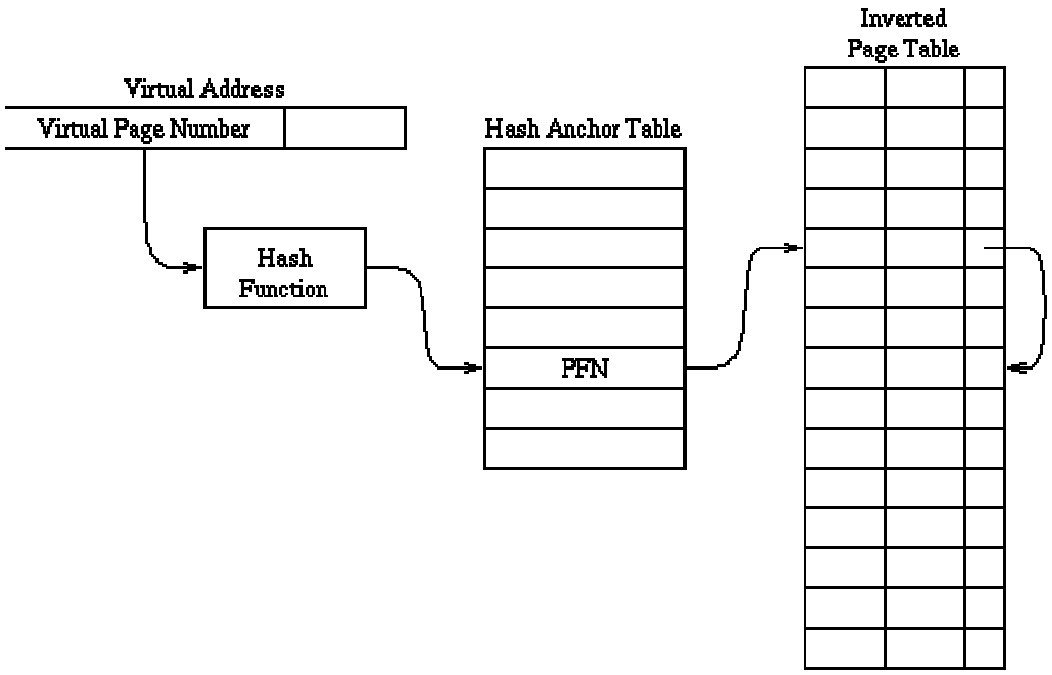

What does this buy us? Sparse address spaces and easier paging

# Example Addressing on a Multilevel Page Table System

- A logical address (on 32-bit x86 with 4k page size) is divided into
  - A page number consisting of 20 bits
  - A page offset consisting of 12 bits
- Divide the page number into
  - A 10-bit page directory
  - A 10-bit page number

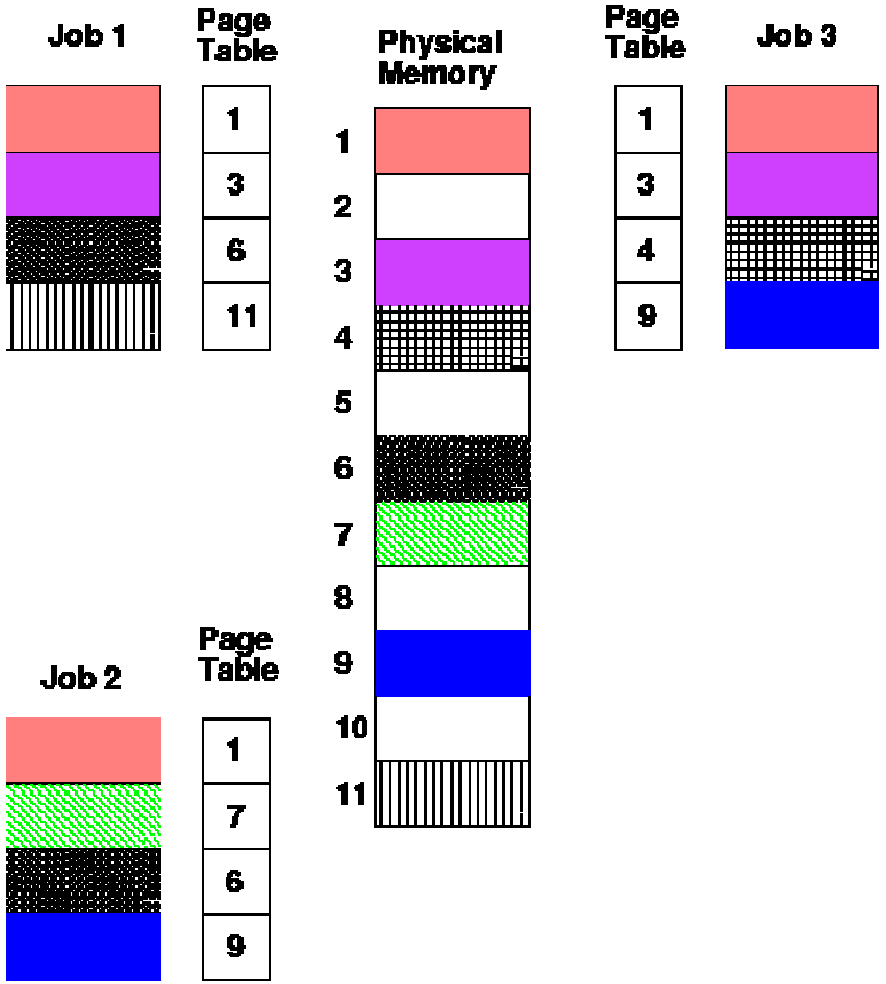


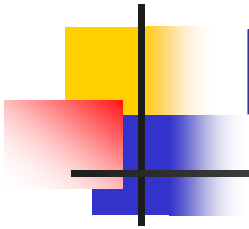
# Inverted Page Tables




- Hash the process ID and virtual page number to get an index into the HAT.
- Look up a Physical Frame Number in the HAT.
- Look at the inverted page table entry, to see if it is the right process ID and virtual page number. If it is, you're done.
- If the PID or VPN does not match, follow the pointer to the next link in the hash chain. Again, if you get a match then you're done; if you don't, then you continue. Eventually, you will either get a match or you will find a pointer that is marked invalid. If you get a match, then you've got the translation; if you get the invalid pointer, then you have a miss.

# Shared Pages

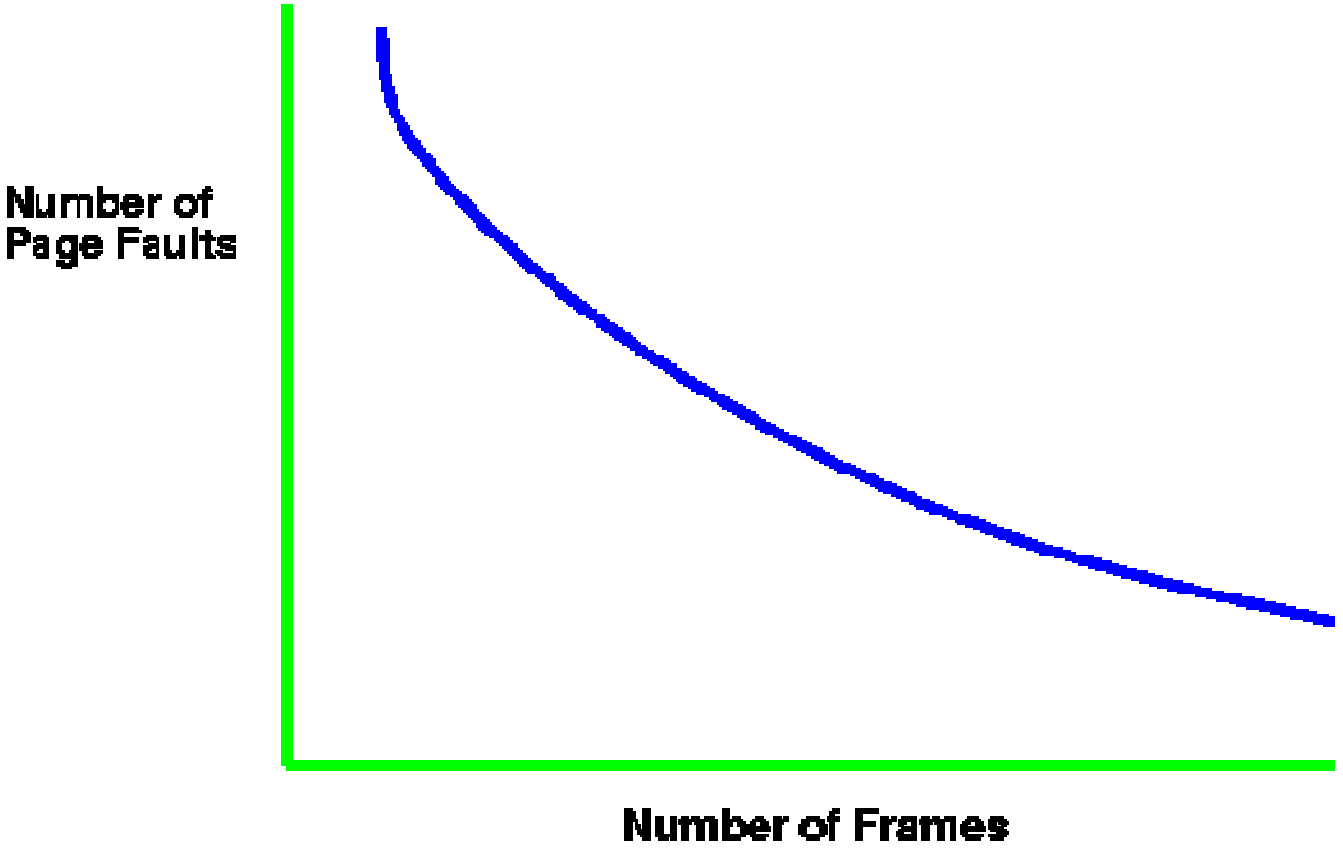




# Principle of Optimality

- Description:
  - Assume that each page can be labeled with the number of instructions that will be executed before that page is first referenced, i.e., we would know the future reference string for a program.
  - Then the optimal page algorithm would choose the page with the highest label to be removed from the memory.
- Impractical because it needs to know future references

# Average Paging Behavior with Increasing Page Frames



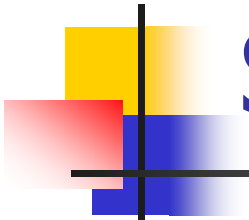
# Belady's Anomaly (for FIFO)

FIFO with 4 physical pages

12 references, 10 faults

As the number of page frames increase, so does the fault rate.

| Page Refs | 4 Page Frames | | | | |
|---|---|---|---|---|---|
| | Fault? | Page Contents | | | |
| A | yes | A | | | |
| B | yes | B | A | | |
| C | yes | C | B | A | |
| D | yes | D | C | B | A |
| A | no | D | C | B | A |
| B | no | D | C | B | A |
| E | yes | E | D | C | B |
| A | yes | A | E | D | C |
| B | yes | B | A | E | D |
| C | yes | C | B | A | E |
| D | yes | D | C | B | A |
| E | yes | E | D | C | B |

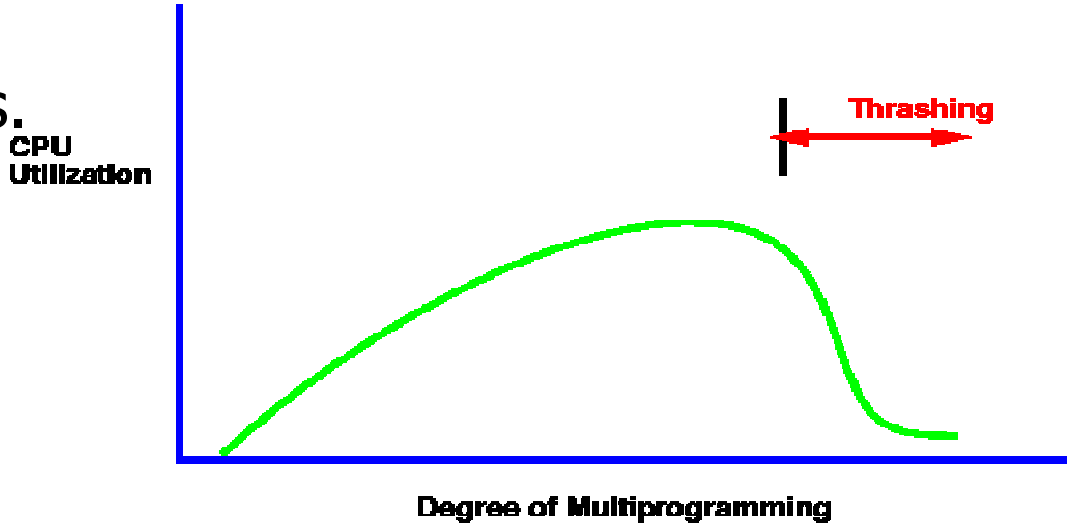


# Second Chance

- Only one reference bit in the page table entry.
  - 0 initially
  - 1 When a page is referenced
- pages are kept in FIFO order using a circular list.
- Choose “victim” to evict
  - Select head of FIFO
  - If page has reference bit set, reset bit and select next page in FIFO list.
  - keep processing until you reach page with zero reference bit and page that one out.
- System V uses a variant of second chance

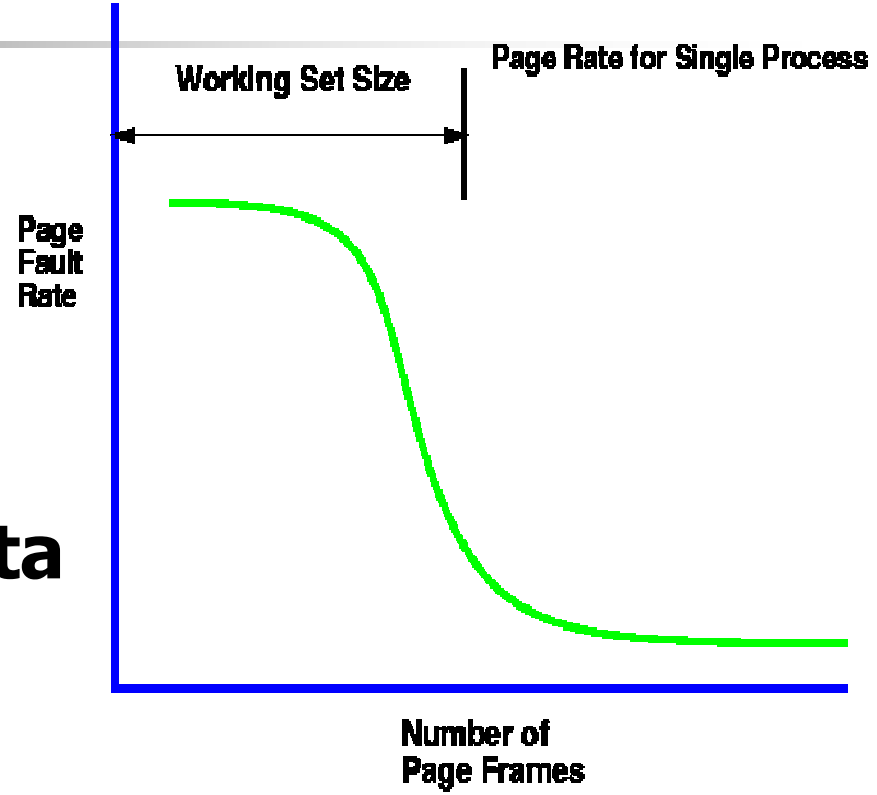
# Thrashing and CPU Utilization

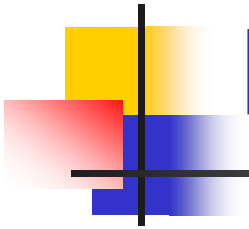
- As the page rate goes up, processes get suspended on page out queues for the disk.
- the system may try to optimize performance by starting new jobs.
- starting new jobs will reduce the number of page frames available to each process, increasing the page fault requests.
- system throughput plunges.



# Working Set

- the working set model assumes locality.
- **the principle of locality states that a program clusters its access to data and text temporally**.
- *As the number of page frames increases above some threshold, the page fault rate will drop dramatically*.





# Page Size Considerations

- Small pages
  - Reason:
    - Locality of reference tends to be small (256)
    - Less fragmentation
  - Problem: require large page tables
- Large pages
  - Reason
    - Small page table
    - I/O transfers have high seek time, so better to transfer more data per seek
  - Problem: Internal fragmentation, needless caching