Recall: Scheduling Policy Goals/Criteria

- **Minimize Response Time**
  - Minimize elapsed time to do an operation (or job)
  - Response time is what the user sees:
    » Time to echo a keystroke in editor
    » Time to compile a program
    » Real-time Tasks: Must meet deadlines imposed by World

- **Maximize Throughput**
  - Maximize operations (or jobs) per second
  - Throughput related to response time, but not identical:
    » Minimizing response time will lead to more context switching than if you only maximized throughput
  - Two parts to maximizing throughput
    » Minimize overhead (for example, context-switching)
    » Efficient use of resources (CPU, disk, memory, etc)

- **Fairness**
  - Share CPU among users in some equitable way
  - Fairness is not minimizing average response time:
    » Better average response time by making system less fair

Recall: What if we Knew the Future?

- Could we always mirror best FCFS?
- **Shortest Job First (SJF):**
  - Run whatever job has the least amount of computation to do
  - Sometimes called “Shortest Time to Completion First” (STCF)
- **Shortest Remaining Time First (SRTF):**
  - Preemptive version of SJF: if job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called “Shortest Remaining Time to Completion First” (SRTCF)
- These can be applied either to a whole program or the current CPU burst of each program
  - 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

Recall: Multi-Level Feedback Scheduling

- **Long-Running Compute Tasks Demoted to Low Priority**
- Another method for exploiting past behavior
  - First used in CTSS
  - Multiple queues, each with different priority
    » Higher priority queues often considered “foreground” tasks
  - Each queue has its own scheduling algorithm
    » e.g. foreground – RR, background – FCFS
    » Sometimes multiple RR priorities with quantum increasing exponentially (highest:1ms, next:2ms, next: 4ms, etc)
- Adjust each job’s priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn’t expire, push up one level (or to top)
Case Study: Linux O(1) Scheduler

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

Linux Completely Fair Scheduler (CFS)

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

O(1) Scheduler Continued

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

CFS (Continued)

- Idea: track amount of “virtual time” received by each process when it is executing
  - Take real execution time, scale by weighting factor
    » higher priority ⇒ real time divided by larger weight
    » Actually – divide by \( \text{current weight}/(\text{sum of all weights}) \)
  - Keep virtual time advancing at same rate
- Targeted latency (\( T_L \)): period of time after which all processes get to run at least a little
  - Each process runs with quantum \( \left( \frac{W_p}{\sum W_i} \right) \times T_L \)
  - Never smaller than “minimum granularity”
- Use of Red-Black tree to hold all runnable processes as sorted on vruntime variable
  - O(log n) time to perform insertions/deletions
    » Cash the item at far left (item with earliest vruntime)
  - When ready to schedule, grab version with smallest vruntime (which will be item at the far left).
CFS Examples

• Suppose Targeted latency = 20ms, Minimum Granularity = 1ms
• Two CPU bound tasks with same priorities
  – Both switch with 10ms
• Two CPU bound tasks separated by nice value of 5
  – Nice values scale weights exponentially: \( \text{Weight} = 1024/(1.25)^{\text{nice}} \)
  – Since \((1.25)^5 \approx 3\), one task gets 5ms, another gets 15ms
• 40 tasks: each gets 1ms (no longer totally fair)
• One CPU bound task, one interactive task same priority
  – While interactive task sleeps, CPU bound task runs and increments vruntime
  – When interactive task wakes up, runs immediately, since it is behind on vruntime
• Group scheduling facilities (2.6.24)
  – Can give fair fractions to groups (like a user or other mechanism for grouping processes)
  – Two users, one starts 1 process, other starts 40, each gets 50% of CPU

Real-Time Scheduling (RTS)

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

Example: Workload Characteristics

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

Example: Round-Robin Scheduling Doesn’t Work
Earliest Deadline First (EDF)

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

\[
T_i = (4,1) \quad T_j = (5,2) \quad T_k = (7,2)
\]

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

A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter?
  - When there aren’t enough resources to go around

- When should you simply buy a faster computer?
  - (Or network link, or expanded highway, or …)
  - One approach: Buy it when it will pay for itself in improved response time
    - Perhaps you’re paying for worse response time in reduced productivity, customer angst, etc…
    - Might think that you should buy a faster X when X is utilized 100%, but usually, response time goes to infinity as utilization \(\rightarrow\) 100%

- An interesting implication of this curve:
  - Most scheduling algorithms work fine in the “linear” portion of the load curve, fail otherwise
  - Argues for buying a faster X when hit “knee” of curve

Administrivia

- Midterm I graded:
  - Mean 60.27, Std Dev: 14.71, Low: 16.25, High: 89.75
  - Regrade requests before Monday 3/11 @ midnight
- Solutions are posted
- Homework 2 due Today!
- Project 1 code due on Friday (3/8)
- Don’t forget to allocate memory for objects!
  - If a structure is declared locally to a procedure, then it will go away when procedure returns!!!
  - Lots of page faults are likely caused by bad memory allocation!

Starvation vs Deadlock

- Starvation: thread waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular waiting for resources
  - Thread A owns Res 1 and is waiting for Res 2
  - Thread B owns Res 2 and is waiting for Res 1

\[
\text{Thread A}
\]

\[
\text{Thread B}
\]

\[
\text{Res 1}
\]

\[
\text{Res 2}
\]
Conditions for Deadlock

- Deadlock not always deterministic – Example 2 mutexes:
  
<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.P();</td>
<td>y.P();</td>
</tr>
<tr>
<td>y.P();</td>
<td>x.P();</td>
</tr>
<tr>
<td>y.V();</td>
<td>x.V();</td>
</tr>
<tr>
<td>x.V();</td>
<td>y.V();</td>
</tr>
</tbody>
</table>
  
  - Deadlock won’t always happen with this code
    » Have to have exactly the right timing (“wrong” timing?)
    » So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant...
- Deadlocks occur with multiple resources
  - Means you can’t decompose the problem
  - Can’t solve deadlock for each resource independently
- Example: System with 2 disk drives and two threads
  - Each thread needs 2 disk drives to function
  - Each thread gets one disk and waits for another one

Bridge Crossing Example

- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
  - Several cars may have to be backed up
- Starvation is possible
  - East-going traffic really fast ⇒ no one goes west

Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    » Protocol: Always go east-west first, then north-south
  - Called “dimension ordering” (X then Y)

Dining Lawyers Problem

- Five chopsticks/Five lawyers (really cheap restaurant)
  - Free-for all: Lawyer will grab any one they can
  - Need two chopsticks to eat
- What if all grab at same time?
  - Deadlock!
- How to fix deadlock?
  - Make one of them give up a chopstick (Hah!)
  - Eventually everyone will get chance to eat
- How to prevent deadlock?
  - Never let lawyer take last chopstick if no hungry lawyer has two chopsticks afterwards
Four requirements for Deadlock

- Mutual exclusion
  - Only one thread at a time can use a resource.
- Hold and wait
  - Thread holding at least one resource is waiting to acquire additional resources held by other threads.
- No preemption
  - Resources are released only voluntarily by the thread holding the resource, after thread is finished with it.
- Circular wait
  - There exists a set \( \{T_1, \ldots, T_n\} \) of waiting threads:
    - \( T_1 \) is waiting for a resource that is held by \( T_2 \)
    - \( T_2 \) is waiting for a resource that is held by \( T_3 \)
    - \( \ldots \)
    - \( T_n \) is waiting for a resource that is held by \( T_1 \)

Resource-Allocation Graph

- System Model
  - A set of Threads \( T_1, T_2, \ldots, T_n \)
  - Resource types \( R_1, R_2, \ldots, R_m \)
  - CPU cycles, memory space, I/O devices
  - Each resource type \( R_i \) has \( W_i \) instances
  - Each thread utilizes a resource as follows:
    - \( \text{Request() / Use() / Release()} \)
- Resource-Allocation Graph:
  - \( V \) is partitioned into two types:
    - \( T = \{T_1, T_2, \ldots, T_n\} \), the set threads in the system.
    - \( R = \{R_1, R_2, \ldots, R_m\} \), the set of resource types in system
  - request edge – directed edge \( T_1 \rightarrow R_j \)
  - assignment edge – directed edge \( R_j \rightarrow T_i \)

Resource-Allocation Graph Examples

- Model:
  - request edge – directed edge \( T_1 \rightarrow R_j \)
  - assignment edge – directed edge \( R_j \rightarrow T_i \)

Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preempting resources and/or terminating tasks
- Ensure that system will never enter a deadlock
  - Need to monitor all lock acquisitions
  - Selectively deny those that might lead to deadlock
- Ignore the problem and pretend that deadlocks never occur in the system
  - Used by most operating systems, including UNIX
  - This is not say that this is a "method", rather intentional ignorance?
### Deadlock Detection Algorithm

- Only one of each type of resource ⇒ look for loops
- More General Deadlock Detection Algorithm
  - Let \([X]\) represent an \(m\)-ary vector of non-negative integers (quantities of resources of each type):
    \[
    \begin{align*}
    \text{[FreeResources]} & : \text{Current free resources each type} \\
    \text{[Request]}_X & : \text{Current requests from thread } X \\
    \text{[Alloc]}_X & : \text{Current resources held by thread } X
    \end{align*}
    \]
  - See if tasks can eventually terminate on their own
    - \([\text{Avail}] = [\text{FreeResources}]\)
    - Add all nodes to UNFINISHED
    - \(\text{do}\{\text{done} = \text{true} \}
    - \text{Foreach node in UNFINISHED \{}\)
      - \(\text{if } ([\text{Request}]_{\text{node}} <= [\text{Avail}]) \{\)
        - \(\text{remove node from UNFINISHED}\)
        - \([\text{Avail}] = [\text{Avail}] + [\text{Alloc}]_{\text{node}}\)
        - \(\text{done} = \text{false}\)
      - \}\}
    - \(\text{until(done)}\)
  - Nodes left in UNFINISHED ⇒ deadlocked

### What to do when detect deadlock?

- Terminate thread, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - Shoot a dining lawyer
  - But, not always possible – killing a thread holding a mutex leaves world inconsistent
- Preempt resources without killing off thread
  - Take away resources from thread temporarily
  - Doesn’t always fit with semantics of computation
- Roll back actions of deadlocked threads
  - Hit the rewind button on TiVo, pretend last few minutes never happened
  - For bridge example, make one car roll backwards (may require others behind him)
  - Common technique in databases (transactions)
  - Of course, if you restart in exactly the same way, may reenter deadlock once again
- Many operating systems use other options

### Techniques for Preventing Deadlock

- Infinite resources
  - Include enough resources so that no one ever runs out of resources. Doesn’t have to be infinite, just large
  - Give illusion of infinite resources (e.g. virtual memory)
  - Examples:
    - Bay bridge with 12,000 lanes. Never wait!
    - Infinite disk space (not realistic yet?)
- No Sharing of resources (totally independent threads)
  - Not very realistic
- Don’t allow waiting
  - How the phone company avoids deadlock
    - Call to your Mom in Toledo, works its way through the phone lines, but if blocked get busy signal.
  - Technique used in Ethernet/some multiprocessor nets
    - Everyone speaks at once. On collision, back off and retry
  - Inefficient, since have to keep retrying
    - Consider: driving to San Francisco; when hit traffic jam, suddenly you’re transported back home and told to retry!

### Techniques for Preventing Deadlock (cont’d)

- Make all threads request everything they’ll need at the beginning.
  - Problem: Predicting future is hard, tend to over-estimate resources
  - Example:
    - If need 2 chopsticks, request both at same time
    - Don’t leave home until we know no one is using any intersection between here and where you want to go; only one car on the Bay Bridge at a time
- Force all threads to request resources in a particular order preventing any cyclic use of resources
  - Thus, preventing deadlock
  - Example (x.P, y.P, z.P,...)
    - Make tasks request disk, then memory, then...
    - Keep from deadlock on freeways around SF by requiring everyone to go clockwise
Review: Train Example (Wormhole-Routed Network)

- Circular dependency (Deadlock!)
  - Each train wants to turn right
  - Blocked by other trains
  - Similar problem to multiprocessor networks
- Fix? Imagine grid extends in all four directions
  - Force ordering of channels (tracks)
    » Protocol: Always go east-west first, then north-south
  - Called “dimension ordering” (X then Y)

Banker’s Algorithm for Preventing Deadlock

- Toward right idea:
  - State maximum (max) resource needs in advance
  - Allow particular thread to proceed if:
    - (available resources - #requested) \geq \text{max}
    - remaining that might be needed by any thread
- Banker’s algorithm (less conservative):
  - Allocate resources dynamically
    » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting
      ((\text{Max}_\text{node}) - \text{[Alloc\_node]} \leq \text{[Avail]}) for ((\text{Request}_\text{node}) \leq \text{[Avail]})
  Grant request if result is deadlock free (conservative!)

```plaintext
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
    done = true
    Foreach node in UNFINISHED {
        if ([Request\_node] \leq [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Alloc\_node]
            done = false
        }
    }
} until(done)
```

» Technique: pretend each request is granted, then run deadlock detection algorithm, substituting
((\text{Max}_\text{node}) - \text{[Alloc\_node]} \leq \text{[Avail]}) for ((\text{Request}_\text{node}) \leq \text{[Avail]})
Grant request if result is deadlock free (conservative!)
Banker’s Algorithm for Preventing Deadlock

• Toward right idea:
  – State maximum resource needs in advance
  – Allow particular thread to proceed if:
    
    \[(\text{available resources} - \#\text{requested}) \geq \text{max remaining that might be needed by any thread}\]
  
• Banker’s algorithm (less conservative):
  – Allocate resources dynamically
    » Evaluate each request and grant if some ordering of threads is still deadlock free afterward
    » Technique: pretend each request is granted, then run deadlock detection algorithm, substituting
      \[(\text{Max node} - \text{Alloc node}) \leq \text{Avail})\]
      for \[(\text{Request node} \leq \text{Avail})\]
      Grant request if result is deadlock free (conservative!)
    » Keeps system in a “SAFE” state, i.e. there exists a sequence \(T_1, T_2, \ldots, T_n\) with \(T_1\) requesting all remaining resources, finishing, then \(T_2\) requesting all remaining resources, etc..
  
• Algorithm allows the sum of maximum resource needs of all current threads to be greater than total resources

Banker’s Algorithm Example

• Banker’s algorithm with dining lawyers
  – “Safe” (won’t cause deadlock) if when try to grab chopstick either:
    » Not last chopstick
    » Is last chopstick but someone will have two afterwards
  – What if k-handed lawyers? Don’t allow if:
    » It’s the last one, no one would have k
    » It’s 2nd to last, and no one would have k-1
    » It’s 3rd to last, and no one would have k-2
    » …

Virtualizing Resources

• Physical Reality: Different Processes/Threads share the same hardware
  – Need to multiplex CPU (Just finished: scheduling)
  – Need to multiplex use of Memory (starting today)
  – Need to multiplex disk and devices (later in term)

• Why worry about memory sharing?
  – The complete working state of a process and/or kernel is defined by its data in memory (and registers)
  – Consequently, cannot just let different threads of control use the same memory
    » Physics: two different pieces of data cannot occupy the same locations in memory
  – Probably don’t want different threads to even have access to each other’s memory if in different processes (protection)

Next Objective

• Dive deeper into the concepts and mechanisms of memory sharing and address translation
• Enabler of many key aspects of operating systems
  – Protection
  – Multi-programming
  – Isolation
  – Memory resource management
  – I/O efficiency
  – Sharing
  – Inter-process communication
  – Debugging
  – Demand paging
• Today: Translation
Recall: Single and Multithreaded Processes

- Threads encapsulate concurrency
  - “Active” component of a process
- Address spaces encapsulate protection
  - Keeps buggy program from trashing the system
  - “Passive” component of a process

Important Aspects of Memory Multiplexing

- Protection:
  - Prevent access to private memory of other processes
    - Different pages of memory can be given special behavior (Read Only, Invisible to user programs, etc).
    - Kernel data protected from User programs
    - Programs protected from themselves
- Controlled overlap:
  - Separate state of threads should not collide in physical memory. Obviously, unexpected overlap causes chaos!
  - Conversely, would like the ability to overlap when desired (for communication)
- Translation:
  - Ability to translate accesses from one address space (virtual) to a different one (physical)
  - When translation exists, processor uses virtual addresses, physical memory uses physical addresses
  - Side effects:
    - Can be used to avoid overlap
    - Can be used to give uniform view of memory to programs

Recall: Loading

- Processor
- Memory
- OS
- Processes
- Files
- Windows
- Sockets
- Hardware
- ISA
- Threads
- Address Spaces

Process view of memory

- data1: dw 32
- start: lw r1,0(data1)
- jal checkit
- loop: addi r1, r1, -1
- bnz r1, loop
- checkit:

Physical addresses

- Assume 4byte words
- 0x300 = 4 * 0x0C0
- 0x0C0 = 0000 1100 0000
- 0x300 = 0011 0000 0000

Binding of Instructions and Data to Memory

- data1: dw 32
- start: lw r1,0(data1)
- jal checkit
- loop: addi r1, r1, -1
- bnz r1, loop
- checkit:

Physical address

- Assume 4byte words
- 0x300 = 0000 1100 0000
- 0x0C0 = 0000 1100 0000
- 0x000200 = 0000 0000 0000
- 0x000200 = 0000 0000 0000
- 0x000200 = 0000 0000 0000
- 0x000200 = 0000 0000 0000
- 0x000200 = 0000 0000 0000
- 0x000200 = 0000 0000 0000
Binding of Instructions and Data to Memory

Process view of memory

Physical addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0000000020</td>
</tr>
<tr>
<td>0x0300</td>
<td>000000020</td>
</tr>
<tr>
<td>0x0900</td>
<td>8C2000C0</td>
</tr>
<tr>
<td>0x0904</td>
<td>0C000820</td>
</tr>
<tr>
<td>0x0908</td>
<td>0280</td>
</tr>
<tr>
<td>0x090C</td>
<td>2021FFFF</td>
</tr>
<tr>
<td>0x0910</td>
<td>14200242</td>
</tr>
<tr>
<td>0x0A00</td>
<td></td>
</tr>
<tr>
<td>0xFFFF</td>
<td></td>
</tr>
</tbody>
</table>

Physical Memory

Second copy of program from previous example

Process view of memory

Physical addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0300</td>
<td>000000020</td>
</tr>
<tr>
<td>0x0900</td>
<td>8C2000C0</td>
</tr>
<tr>
<td>0x0904</td>
<td>0C000820</td>
</tr>
<tr>
<td>0x0908</td>
<td>0280</td>
</tr>
<tr>
<td>0x090C</td>
<td>2021FFFF</td>
</tr>
<tr>
<td>0x0910</td>
<td>14200242</td>
</tr>
<tr>
<td>0x0A00</td>
<td></td>
</tr>
<tr>
<td>0xFFFF</td>
<td></td>
</tr>
</tbody>
</table>

Physical Memory

Need address translation!

Multi-step Processing of a Program for Execution

• Preparation of a program for execution involves components at:
  – Compile time (i.e., “gcc”)
  – Link/Load time (UNIX “ld” does link)
  – Execution time (e.g., dynamic libs)

• Addresses can be bound to final values anywhere in this path
  – Depends on hardware support
  – Also depends on operating system

• Dynamic Libraries
  – Linking postponed until execution
  – Small piece of code, stub, used to locate appropriate memory-resident library routine
  – Stub replaces itself with the address of the routine, and executes routine

One of many possible translations!
Where does translation take place?
  Compile time, Link/Load time, or Execution time?
Recall: Uniprogramming

- Uniprogramming (no Translation or Protection)
  - Application always runs at the same place in physical memory since only one application at a time
  - Application can access any physical address
  - Application given illusion of dedicated machine by giving it reality of a dedicated machine

Recall: General Address translation

- Address Space:
  - All the addresses and state a process can touch
  - Each process and kernel has different address space

  Consequently, two views of memory:
  - View from the CPU (what program sees, virtual memory)
  - View from memory (physical memory)

- Translation makes it much easier to implement protection
  - If task A cannot even gain access to task B’s data, no way for A to adversely affect B

- With translation, every program can be linked/loaded into same region of user address space

Multiprogramming (primitive stage)

- Multiprogramming without Translation or Protection
  - Must somehow prevent address overlap between threads

  - Use Loader/Linker: Adjust addresses while program loaded into memory (loads, stores, jumps)
    » Everything adjusted to memory location of program
    » Translation done by a linker-loader (relocation)
    » Common in early days (… till Windows 3.x, 95?)

  - With this solution, no protection: bugs in any program can cause other programs to crash or even the OS

Multiprogramming (Version with Protection)

- Can we protect programs from each other without translation?

  - Yes: use two special registers BaseAddr and LimitAddr to prevent user from straying outside designated area
    » If user tries to access an illegal address, cause an error
  - During switch, kernel loads new base/limit from PCB (Process Control Block)
    » User not allowed to change base/limit registers
Summary

- **Linux CFS Scheduler**
  - Fair fraction of CPU to threads, modulated by priority
  - Approximates an “ideal” multitasking processor
- **Real-time scheduling**
  - Need to meet a deadline, predictability essential
  - Earliest Deadline First (EDF) and Rate Monotonic (RM) scheduling
- **Starvation vs. Deadlock**
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
- **Four conditions for deadlocks**
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- **Techniques for addressing Deadlock**
  - Allow system to enter deadlock and then recover
  - Ensure that system will never enter a deadlock
  - Ignore the problem and pretend that deadlocks never occur