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

- 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)
Real-Time Scheduling (RTS)

- Efficiency is important but **predictability** is essential:
  - We need to be able to predict with confidence the worst case response times for systems
  - In RTS, performance guarantees are:
    » Task- and/or class centric
    » Often ensured a priori
  - In conventional systems, performance is:
    » System oriented and often throughput oriented
    » Post-processing (... wait and see ...)
  - Real-time is about enforcing predictability, and does not equal to 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
- Times have deadlines (D) and known computation times (C)
- Example Setup:

Example: Round-Robin Scheduling Doesn't Work

- Tasks periodic with period P and computation C in each period: (P, C)
- Preemptive priority-based dynamic scheduling
- Each task is assigned a (current) priority based on how close the absolute deadline is.
- The scheduler always schedules the active task with the closest absolute deadline.
EDF: Schedulability Test

Theorem (Utilization-based Schedulability Test):
A task set \( T_1, T_2, \ldots, T_n \) with \( D_i = P_i \) is schedulable by the earliest deadline first (EDF) scheduling algorithm if

\[
\sum_{i=1}^{n} \left( \frac{C_i}{D_i} \right) \leq 1
\]

Exact schedulability test (necessary + sufficient)
Proof: [Liu and Layland, 1973]

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
    » Assuming 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 coming up soon!
  - Wednesday 10/14, 6:30-9:30pm
  - In 145/155 Dwinelle
  - No class that day, extra office hours
• Details
  - Intend this to be a 2-hour exam in 3 hour slot
  - 1 page of hand-written notes, both sides
  - Closed book
  - Topics will include the material from that Monday

Starvation vs Deadlock

• 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

  - Deadlock \n Starvation but not vice versa
    » Starvation can end (but doesn't have to)
    » Deadlock can't end without external intervention
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 the 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:
    - \( \text{Request}() / \text{Use}() / \text{Release}() \)

- **Resource-Allocation Graph**:
  - \( V \) is partitioned into two types:
    - \( \forall T = \{ T_1, T_2, \ldots, T_n \} \), the set of threads in the system.
    - \( \forall R = \{ R_1, R_2, \ldots, R_m \} \), the set of resource types in system.
  - request edge – directed edge \( T_i \rightarrow R_j \)
  - assignment edge – directed edge \( R_j \rightarrow T_i \)

### Resource Allocation Graph Examples

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

- **Simple Resource Allocation Graph**
- **Allocation Graph With Deadlock**
- **Allocation Graph With Cycle, but No Deadlock**

### Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preempts 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
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):
    - \([\text{FreeResources}]\): Current free resources each type
    - \([\text{Request}_X]\): Current requests from thread \(X\)
    - \([\text{Alloc}_X]\): Current resources held by thread \(X\)
  - See if tasks can eventually terminate on their own
    - \([\text{Avail}] = [\text{FreeResources}]\)
    - Add all nodes to UNFINISHED
    - do {
      - done = true
      - Foreach node in UNFINISHED {
        - if \(([\text{Request}_\text{node}] \leq [\text{Avail}])\) {
          - remove node from UNFINISHED
          - \([\text{Avail}] = [\text{Avail}] + [\text{Alloc}_\text{node}]\)
          - done = false
        }
      }
    } until(done)
  - Nodes left in UNFINISHED ⇒ deadlocked

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!

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 (con’t)

- 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 (\(x.P, y.P, z.P, \ldots\))
    - 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 resource needs in advance
  - Allow particular thread to proceed if:
    - (available resources - #requested) ≥ 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
      - ([Maxnode] - [Allocnode] ≤ [Avail]) for ([Requestnode] ≤ [Avail])
    - Grant request if result is deadlock free (conservative!)
    - Keeps system in a "SAFE" state, i.e. there exists a sequence {T1, T2, … Tn} with T1 requesting all remaining resources, finishing, then T2 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 (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 (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: Linking, Segmentation, Paged Virtual Address

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

Recall: Loading

Important Aspects of Memory Multiplexing

- 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
- 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
Binding of Instructions and Data to Memory

Process view of memory

Physical addresses

Assume 4-byte words:
- $0x300 = 4 \times 0xOC0$
- $0x0C0 = 0000 \ 1100 \ 0000$
- $0x300 = 0011 \ 0000 \ 0000$

Second copy of program from previous example

Need address translation!

One of many possible translations!
- Where does translation take place?
  - Compile time
  - Link/Load time
  - Execution time?
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

Recall: Uniprogramming

- Uniprogramming (no Translation or Protection)
  - Application always runs at 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 reality of a dedicated machine

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
Recall: General Address translation

- Recall: 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 box (MMU) converts between the two views
- 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

Simple Example: Base and Bounds (CRAY-1)

- Could use base/limit for dynamic address translation - translation happens at execution:
  - Alter address of every load/store by adding “base”
  - Generate error if address bigger than limit
- This gives program the illusion that it is running on its own dedicated machine, with memory starting at 0
  - Program gets continuous region of memory
  - Addresses within program do not have to be relocated when program placed in different region of DRAM

Issues with Simple B&B Method

- Fragmentation problem
  - Not every process is the same size
  - Over time, memory space becomes fragmented
- Missing support for sparse address space
  - Would like to have multiple chunks/program
  - E.g.: Code, Data, Stack
- Hard to do inter-process sharing
  - Want to share code segments when possible
  - Want to share memory between processes
  - Helped by providing multiple segments per process

More Flexible Segmentation

- Logical View: multiple separate segments
  - Typical: Code, Data, Stack
  - Others: memory sharing, etc
  - Each segment is given region of contiguous memory
  - Has a base and limit
  - Can reside anywhere in physical memory
Implementation of Multi-Segment Model

- Segment map resides in processor
  - Segment number mapped into base/limit pair
  - Base added to offset to generate physical address
  - Error check catches offset out of range
- As many chunks of physical memory as entries
  - Segment addressed by portion of virtual address
  - However, could be included in instruction instead:
    » x86 Example: mov [es:bx],ax.
- What is “V/N” (valid / not valid)?
  - Can mark segments as invalid; requires check as well

Example: Four Segments (16 bit addresses)

<table>
<thead>
<tr>
<th>Seg ID #</th>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (code)</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1 (data)</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2 (shared)</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3 (stack)</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>

Intel x86 Special Registers

Intel x86 Special Registers

Example: Four Segments (16 bit addresses)

Running more programs than fit in memory: Swapping

- Q: What if not all processes fit in memory?
- A: Swapping: Extreme form of Context Switch
  - In order to make room for next process, some or all of the previous process is moved to disk
  - This greatly increases the cost of context-switching

- Desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory
Problems with Segmentation

- Must fit variable-sized chunks into physical memory
- May move processes multiple times to fit everything
- Limited options for swapping to disk
- Fragmentation: wasted space
  - External: free gaps between allocated chunks
  - Internal: don't need all memory within allocated chunks

Recall: General Address Translation

Paging: Physical Memory in Fixed Size Chunks

- Solution to fragmentation from segments?
  - Allocate physical memory in fixed size chunks ("pages")
  - Every chunk of physical memory is equivalent
    » Can use simple vector of bits to handle allocation:
      00110001110001101 ... 110010
    » Each bit represents page of physical memory
      1 => allocated, 0 => free

- Should pages be as big as our previous segments?
  - No: Can lead to lots of internal fragmentation
    » Typically have small pages (1K-16K)
  - Consequently: need multiple pages/segment

How to Implement Paging?

- Page Table (One per process)
  - Resides in physical memory
  - Contains physical page and permission for each virtual page
    » Permissions include: Valid bits, Read, Write, etc
  - Virtual address mapping
    » Offset from Virtual address copied to Physical Address
    » Example: 10 bit offset \( \Rightarrow \) 1024-byte pages
    » Virtual page # is all remaining bits
      » Example for 32-bits: 32-10 = 22 bits, i.e. 4 million entries
      » Physical page # copied from table into physical address
  - Check Page Table bounds and permissions
Simple Page Table Example

Example (4 byte pages)

Virtual Memory

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0000</td>
</tr>
<tr>
<td>0x04</td>
<td>0001</td>
</tr>
<tr>
<td>0x08</td>
<td>0002</td>
</tr>
<tr>
<td>0x0C</td>
<td>0003</td>
</tr>
</tbody>
</table>

Physical Memory

<table>
<thead>
<tr>
<th>Physical Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
</tr>
</tbody>
</table>

Virtual Address (Process A):

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td>V,R</td>
</tr>
<tr>
<td>page #1</td>
<td>V,R</td>
</tr>
<tr>
<td>page #2</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #3</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #4</td>
<td>N</td>
</tr>
<tr>
<td>page #5</td>
<td>V,R,W</td>
</tr>
</tbody>
</table>

Virtual Address (Process B):

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>page #0</td>
<td>V,R</td>
</tr>
<tr>
<td>page #1</td>
<td>N</td>
</tr>
<tr>
<td>page #2</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #3</td>
<td>N</td>
</tr>
<tr>
<td>page #4</td>
<td>V,R,W</td>
</tr>
<tr>
<td>page #5</td>
<td>V,R,W</td>
</tr>
</tbody>
</table>

What about Sharing?

This physical page appears in address space of both processes

E.g., Linux 32-bit

Virtual memory view

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1111</td>
<td>stack</td>
</tr>
<tr>
<td>1111 0000</td>
<td></td>
</tr>
<tr>
<td>1110 1110</td>
<td></td>
</tr>
<tr>
<td>1110 1101</td>
<td></td>
</tr>
<tr>
<td>1110 1011</td>
<td></td>
</tr>
<tr>
<td>1110 0111</td>
<td></td>
</tr>
<tr>
<td>1110 0011</td>
<td></td>
</tr>
<tr>
<td>1110 0001</td>
<td></td>
</tr>
<tr>
<td>1110 0010</td>
<td></td>
</tr>
<tr>
<td>1110 0101</td>
<td></td>
</tr>
<tr>
<td>1110 0100</td>
<td></td>
</tr>
<tr>
<td>1110 0000</td>
<td></td>
</tr>
<tr>
<td>1110 0110</td>
<td></td>
</tr>
<tr>
<td>1110 0100</td>
<td></td>
</tr>
<tr>
<td>1110 0010</td>
<td></td>
</tr>
<tr>
<td>1110 0000</td>
<td></td>
</tr>
</tbody>
</table>

Stack

Memory Mapping Segment

Program segment

Heap

Data segment

Text segment (ELF)

Physical memory view

<table>
<thead>
<tr>
<th>Physical Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>code</td>
</tr>
<tr>
<td>0000 0100</td>
<td></td>
</tr>
<tr>
<td>0000 0110</td>
<td></td>
</tr>
<tr>
<td>0000 0111</td>
<td></td>
</tr>
<tr>
<td>0000 0101</td>
<td></td>
</tr>
<tr>
<td>0000 0100</td>
<td></td>
</tr>
<tr>
<td>0000 0010</td>
<td></td>
</tr>
<tr>
<td>0000 0001</td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
</tr>
<tr>
<td>0001 0000</td>
<td></td>
</tr>
<tr>
<td>0001 0100</td>
<td></td>
</tr>
<tr>
<td>0001 0110</td>
<td></td>
</tr>
<tr>
<td>0001 0101</td>
<td></td>
</tr>
<tr>
<td>0001 0010</td>
<td></td>
</tr>
<tr>
<td>0001 0001</td>
<td></td>
</tr>
<tr>
<td>0001 0000</td>
<td></td>
</tr>
</tbody>
</table>

Summary: Paging

Virtual memory view

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1111 1111</td>
<td>stack</td>
</tr>
<tr>
<td>1111 0000</td>
<td></td>
</tr>
<tr>
<td>1110 1110</td>
<td></td>
</tr>
<tr>
<td>1110 1101</td>
<td></td>
</tr>
<tr>
<td>1110 1011</td>
<td></td>
</tr>
<tr>
<td>1110 0111</td>
<td></td>
</tr>
<tr>
<td>1110 0011</td>
<td></td>
</tr>
<tr>
<td>1110 0001</td>
<td></td>
</tr>
<tr>
<td>1110 0110</td>
<td></td>
</tr>
<tr>
<td>1110 0101</td>
<td></td>
</tr>
<tr>
<td>1110 0100</td>
<td></td>
</tr>
<tr>
<td>1110 0010</td>
<td></td>
</tr>
<tr>
<td>1110 0000</td>
<td></td>
</tr>
<tr>
<td>1110 0111</td>
<td></td>
</tr>
<tr>
<td>1110 0101</td>
<td></td>
</tr>
<tr>
<td>1110 0011</td>
<td></td>
</tr>
<tr>
<td>1110 0001</td>
<td></td>
</tr>
<tr>
<td>1110 0000</td>
<td></td>
</tr>
</tbody>
</table>

Physical memory view

<table>
<thead>
<tr>
<th>Physical Address</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>code</td>
</tr>
<tr>
<td>0000 0100</td>
<td></td>
</tr>
<tr>
<td>0000 0110</td>
<td></td>
</tr>
<tr>
<td>0000 0111</td>
<td></td>
</tr>
<tr>
<td>0000 0101</td>
<td></td>
</tr>
<tr>
<td>0000 0100</td>
<td></td>
</tr>
<tr>
<td>0000 0010</td>
<td></td>
</tr>
<tr>
<td>0000 0001</td>
<td></td>
</tr>
<tr>
<td>0000 0000</td>
<td></td>
</tr>
<tr>
<td>0001 0000</td>
<td></td>
</tr>
<tr>
<td>0001 0100</td>
<td></td>
</tr>
<tr>
<td>0001 0110</td>
<td></td>
</tr>
<tr>
<td>0001 0101</td>
<td></td>
</tr>
<tr>
<td>0001 0010</td>
<td></td>
</tr>
<tr>
<td>0001 0001</td>
<td></td>
</tr>
<tr>
<td>0001 0000</td>
<td></td>
</tr>
</tbody>
</table>

http://static.duarteis.org/img/blogPosts/linuxFlexibleAddressSpaceLayout.png
Summary: Paging

Virtual memory view

Physical memory view

Virtual memory view

Physical memory view

What happens if stack grows to 1110 0000?

 Allocate new pages where room!

Page Table Discussion

• What needs to be switched on a context switch?
  - Page table pointer and limit

• Analysis
  - Pros
    » Simple memory allocation
    » Easy to Share
  - Cons: What if address space is sparse?
    » E.g. on UNIX, code starts at 0, stack starts at \(2^{31} - 1\).
    » With 1K pages, need 2 million page table entries!
  - Cons: What if table really big?
    » Not all pages used all the time ⇒ would be nice to have working set of page table in memory

• How about combining paging and segmentation?

Next time: Multi-level Page Table

Two-level Page Tables

32-bit address:

• Page: a unit of memory translatable by memory management unit (MMU)
  - Typically 1K – 8K
• Page table structure in memory
  - Each user has different page table
  - Address Space switch: change pointer to base of table (hardware register)
  - Hardware traverses page table (for many architectures)
  - MIPS uses software to traverse table
Summary

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
  - Deadlock: circular waiting for resources
- Four conditions for deadlocks
  - 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 threads
  - Circular wait
    » ∃ set \( \{ T_1, ..., T_n \} \) of threads with a cyclic waiting pattern
- 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 in the system

Summary (2)

- Memory is a resource that must be multiplexed
  - Controlled Overlap: only shared when appropriate
  - Translation: Change virtual addresses into physical addresses
  - Protection: Prevent unauthorized sharing of resources
- Simple Protection through segmentation
  - Base + Limit registers restrict memory accessible to user
  - Can be used to translate as well
- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Offset of virtual address same as physical address