Recall: SRTF vs RR Example

- SRTF: Disk Utilization: 9/201 ~ 4.5%
- RR: Disk Utilization: ~90% but lots of wakeups!

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)

Recall: Linux Completely Fair Scheduler (CFS)

- First appeared in 2.6.23, modified in 2.6.24
- 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
- Idea: track amount of “virtual time” received by each process when it is executing
  - Take real execution time, scale by factor to reflect time it would have gotten on ideal multiprocessor
    » So, for instance, multiply real time by N
  - Keep virtual time for every process advancing at same rate
    » Time sliced to achieve multiplexing
  - Uses a red-black tree to always find process which has gotten least amount of virtual time
- Automatically track interactivity:
  - Interactive process runs less frequently ⇒ lower registered virtual time ⇒ will run immediately when ready to run
Recall: Real-Time Scheduling (RTS)

- Efficiency is important but predictability is essential:
  - 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
  - Important for multimedia applications
  - CBS (Constant Bandwidth Server)

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->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

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 ⇒ 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:
  - Thread A
    - x.P();
    - y.P();
    - y.V();
    - x.V();

  - Thread B
    - y.P();
    - x.P();
    - x.V();
    - y.V();

- 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 \( T_3 \)
    » …
    » \( T_n \) 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:
    » 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 $T_i \rightarrow R_j$
  - 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

**Resource Allocation Graph Examples**

- **Recall**:
  - request edge - directed edge $T_i \rightarrow R_j$
  - assignment edge - directed edge $R_j \rightarrow T_i$

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

**Administrivia**

- **Midterm I coming up in 1.5 weeks!**
  - March 11th, 7:00-10:00PM
  - Rooms: 1 PIMENTEL; 2060 VALLEY LSB
  - All topics up to and including next Monday
  - Closed book
  - 1 page hand-written notes both sides
- **HW3 moved 1 week**
  - Sorry about that, we had a bit of a scheduling snafu
Deadlock Detection Algorithm

- Only one of each type of resource $\Rightarrow$ 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 $\Rightarrow$ 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 ($x.P, y.P, z.P,...$)
    - 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 channels is still deadlock free afterward
    » Technique: pretend each request is granted, then run
deadlock detection algorithm, substituting
  \([\text{Max}_{\text{node}} - \text{Alloc}_{\text{node}}] \leq [\text{Avail}]\) for
  \([\text{Request}_{\text{node}}] \leq [\text{Avail}]\)
  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

- "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

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

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

Data addresses

0x0300 00000020
0x0900 8C2000C0
0x0904 0000280
0x0908 2021FFFF
0x090C 14200242
...
0x0A00

Physical addresses

0x0300 00000020
0x0900 8C2000C0
0x0904 0000280
0x0908 2021FFFF
0x090C 14200242
...
0x0A00

Process view of memory

Second copy of program from previous example

Process view of memory

data1: dw 32
start: lw r1,0(data1)
jal checkit
loop: addi r1, r1, -1
bnz r1, r0, loop
checkit: ...

Data addresses

0x1300 00000020
0x1900 8C2004C0
0x1904 0000680
0x1908 2021FFFF
0x190C 14200642
...
0x1A00

Physical addresses

0x1300 00000020
0x1900 8C2004C0
0x1904 0000680
0x1908 2021FFFF
0x190C 14200642
...
0x1A00

• One of many possible translations!
• Where does translation take place?
  Compile time, Link/Load time, or 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

Dynamic Libraries

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 it 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
  - During switch, kernel loads new base/limit from PCB (Process Control Block)
    - User not allowed to change base/limit registers
Better Solution: 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 box (MMU) converts between the two views
- Translation essential to implementing 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

Recall: General Address Translation

Translation Map 1

Translation Map 2

Simple 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

Intel x86 Special Registers

Typical Segment Register Current Priority is RPL of Code Segment (CS)

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</td>
<td>0x4000</td>
<td>0x0800</td>
</tr>
<tr>
<td>1</td>
<td>0x4800</td>
<td>0x1400</td>
</tr>
<tr>
<td>2</td>
<td>0xF000</td>
<td>0x1000</td>
</tr>
<tr>
<td>3</td>
<td>0x0000</td>
<td>0x3000</td>
</tr>
</tbody>
</table>
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

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)
  - Consequentially: 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 ➔ 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

**Virtual Memory**
- `0x00` to `0x04`: `abcd`
- `0x04` to `0x08`: `efgh`
- `0x08` to `0x0C`: `ijkl`

**Page Table**
- `0x00`: `0000 1110` => `0000 1110`
- `0x04`: `0000 1111` => `0001 0100`
- `0x08`: `0000 0000` => `0000 0100`
- `0x0C`: `0000 0111` => `0000 0100`

**Physical Memory**
- `0x00`: `abcd`
- `0x04`: `efgh`
- `0x08`: `ijkl`
- `0x0C`: `abcd`

---

### What about Sharing?

**Virtual Address (Process A):**
- `PageTablePtrA`
- `PageTablePtrB`

**Virtual Address (Process B):**
- `Virtual Page #`  `Offset`

**Shared Page**
- `This physical page appears in address space of both processes`
Summary: Paging

Virtual memory view

Stack: 1111 1111
Heap: 1110 0000
Data: 0100 0000
Code: 1000 0000

Physical memory view

Stack: 1110 0000
Heap: 0111 0000
Data: 0101 0000
Code: 0010 0000

Page Table

0000 0000
0001 0000
0100 0000
0110 0000
1000 0000
1010 0000
1100 0000
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
    - \( \exists \) set \( \{ T_1, \ldots, T_n \} \) 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