CS162
Operating Systems and
Systems Programming
Lecture 11

Scheduling (Finished),
Deadlock, Address Translation

October 5th, 2015
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.
Recall: SRTF Example continued:

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

**SRTF**
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 a 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
  - Minimize miss ratio / maximize completion ratio (firm real-time)
  - 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 $\Rightarrow 100$

- 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
Resources

• Resources - passive entities needed by threads to do their work
  - CPU time, disk space, memory

• Two types of resources
  - Preemptable - can take it away
    » CPU
  - Non-preemptable - must leave it with the thread
    » Disk space, plotter, chunk of virtual address space
    » Mutual exclusion - the right to enter a critical section

• Resources may require exclusive access or may be sharable
  - Read-only files are typically sharable
  - Printers are not sharable during time of printing

• One of the major tasks of an operating system is to manage resources
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:

<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
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, ..., 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 $T_1 \rightarrow R_j$
  - 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 $R_j \rightarrow T_i$

### Diagrams

1. **Simple Resource Allocation Graph**
   - $T_1$ and $T_2$ request resources $R_1$ and $R_2$, respectively.
   - $R_1$ and $R_2$ are assigned to $T_1$ and $T_2$, respectively.

2. **Allocation Graph With Deadlock**
   - $T_3$ requests $R_3$,
     - $R_3$ is already occupied by $T_2$,
     - $T_2$ requests $R_4$, but $R_4$ is already occupied by $T_1$.

3. **Allocation Graph With Cycle, but No Deadlock**
   - A cycle exists without any resource deadlock,
     - $T_5$ requests $R_5$,
     - $R_5$ is assigned to $T_5$.

---

10/5/15
Kubiatowicz CS162 ©UCB Fall 2015
Methods for Handling Deadlocks

- Allow system to enter deadlock and then recover
  - Requires deadlock detection algorithm
  - Some technique for forcibly preemtping 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 $\Rightarrow$ look for loops
- More General Deadlock Detection Algorithm

  - Let $[X]$ represent an m-array 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
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 (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 city center by requiring everyone to go clockwise
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
      
    - 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 (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
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
Recall: Loading

Software
OS Hardware Virtualization

Hardware
ISA

Processor

Threads
Address Spaces
Processes
Windows
Files
Sockets

Memory

Protection Boundary

Networks

Inputs

Displays

Storage

Ctrlr

Memory

OS
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: ...

Physical addresses

0x0300 00000020
... ... 
0x0900 8C2000C0
0x0904 0C002800
0x0908 2021FFFF
0x090C 14200242
... 
0xA000

Physical Memory

0x0000 00000020
0x0300 8C2000C0
0x0900 0C000340
0x0904 2021FFFF
0x0908 14200242
0xFFFF 0x0300 0x0000 00000020

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: ...

Physical addresses

<table>
<thead>
<tr>
<th>Physical addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x300 00000020</td>
</tr>
<tr>
<td>0x900 8C2000C0</td>
</tr>
<tr>
<td>0x904 0C000280</td>
</tr>
<tr>
<td>0x908 2021FFFF</td>
</tr>
<tr>
<td>0x90C 14200242</td>
</tr>
<tr>
<td>0xA00</td>
</tr>
</tbody>
</table>

Need address translation!
Second copy of program from previous example

• 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
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
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
Example of General Address Translation

Translation Map 1

Translation Map 2

Physical 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
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, \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