CS 3723 Operating Systems: Midterm II - Review

Instructor: Dr. Tongping Liu

Thank Dr. Dakai Zhu and Dr. Palden Lama for providing their slides.

Memory Management: Outline

- Background
- Swapping
- Contiguous Memory Allocation and Fragmentation
- Paging
- Structure of the Page Table
- TLB

Simple one: Base and Limit Registers

- Memory protection is required to ensure correct operation
- A pair of base and limit registers define the logical address space of a process. Every memory access is checked by hardware. Any problem? Too slow

Logical vs. Physical Address Space

- Logical address
  - generated by the CPU; also referred to as virtual address
- Physical address
  - address seen by the memory unit
  - Logical and physical addresses are the same in compile-time and load-time address-binding schemes;
- Logical (virtual) and physical addresses differ in execution-time address-binding scheme
  - The mapping from logical address to physical address is done by a hardware called a memory management unit (MMU).
  - We will mainly study how this mapping is done and what hardware support is needed

Memory-Management Unit (MMU)

- Hardware device that maps logical (virtual) address to physical address
- In a simple MMU, the value in the relocation register (base) is added to every address generated by a user process at the time it is sent to memory
- The user program deals with logical addresses, not real physical addresses

Logical addresses (in the range 0~max) and physical addresses (in the range R+0 to R+max for a base value R).

- The user program generates only logical addresses and thinks that the process runs in locations 0 to max.
- logical addresses must be mapped to physical addresses before they are used
Dynamic Loading

- Why dynamic loading?
  - Without this, the size of a process is limited to that of physical memory
- Dynamic loading:
  - Dynamically load routines when they are called
  - All other routines are kept on disk in a loadable format
  - Better memory-space utilization since unused routines are never loaded
  - Useful when large amounts of code are needed to handle infrequently occurring cases like error handling

Dynamic Linking and Shared Libraries

- Static linking
  - System language and library routines are included in the binary code
- Dynamic linking: similar to dynamic loading
  - Linking postponed until execution time
  - Without that, every library will have a copy in the executable file, wasting both disk space and memory
  - A stub is used to locate and load the appropriate memory-resident library routine, when the routine is not existing
  - If it is existing, no need for loading again (PLT)
- Dynamic linking is particularly useful for libraries (one copy, transparent updates)

Swapping

Consider a multi-programming environment:
- Each program must be in the memory to be executed
- Processes come into memory and leave memory when execution is completed

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- Backing store – large enough to accommodate copy of all memory images for all users; must provide direct access to these memory images
- Roll out, roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Swapping can free up memory for additional processes

Contiguous Allocation

- Main memory is usually divided into two partitions:
  - Resident operating system, usually held in low memory
  - User processes, usually held in high memory
- Relocation registers are used to protect user processes from each other, and from changing operating-system code and data
  - MMU maps logical address to physical addresses dynamically
  - But the physical addresses should be contiguous

Contiguous Allocation (Cont)

- Multiple-partition allocation
  - Hole – block of available memory;
    - Holes of various size are scattered throughout memory
    - When a process arrives, OS allocates memory from a hole large enough to accommodate it
    - Operating system maintains information about: a) allocated partitions  b) free partitions (holes)
Dynamic Storage-Allocation Problem

How to satisfy a request of size $n$ from a list of free holes

- **First-fit**: Allocate the first hole that is big enough
- **Best-fit**: Allocate the smallest hole that is big enough; Must search entire list, unless ordered by size
- **Worst-fit**: Allocate the largest hole; Must also search entire list

First-fit and best-fit are better than worst-fit in terms of speed and storage utilization. But all suffer from fragmentation.

Fragmentation

- **External Fragmentation**: Total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation**: Allocated memory may be slightly larger than requested memory; size difference is internal fragmentation

How can we reduce external fragmentation

- **Compaction**: Move memory to place all free memory together in one large block, possible only if relocation is dynamic and is done at execution time

IO problem $\rightarrow$ large overhead

Virtual and Physical Addresses

- Virtual address space
  - Determined by instruction width
  - Same for all processes
- Physical memory indexed by physical addresses
  - Limited by bus size (# of bits)
  - Amount of available memory

**Paging**: A memory-management scheme that permits address space of process to be non-continuous.

Address Translation

- Suppose the logical address space is $2^m$ and page size is $2^n$, so the number of pages is $2^m/2^n$, which is $2^{m-n}$
- Logical Address ($m$ bits) is divided into:
  - **Page number ($p$)** – used as an index into a page table which contains base address of each page in physical memory
  - **Page offset ($d$)** – combined with base address to define the physical memory address that is sent to the memory unit

```
  page number | page offset
  --------    | -------
  p         | d
```

Another Example

- Example:
  - 64 KB virtual memory
  - 32 KB physical memory
  - 4 KB page/frame size $\rightarrow$ 12 bits as offset ($d$)

```
  Address Translation
  Page #3-bits Offset: 12 bits
  Virtual address: 16 bits

  How many virtual pages?
  Frame #3-bits Offset: 12 bits
  Physical address: 15 bits
  How many physical frames?
```
Virtual address

Two-Level Page Tables of Linux

Address Translation Architecture

Page Table Entry: Protection & Other Bits

Page Table Entry (PTE): Information Needed

- **Valid** bit attached for memory protection
  - "valid" indicates that the associated page is a legal page that is in memory
  - "invalid" indicates that the page is not in memory (or not legal for protection)
- Referenced bit: set if the page has been accessed recently
- Dirty (modified) bit: set if data on the page has been modified
- Protection information: read-only/writable/executable or not

Size of each PTE: frame number plus several bits

- Number of bits: power of 2 \( \rightarrow \) if needs 24 bits, use 32b (48)

Two-Level Page Tables of Linux

Solution: multi-level page tables

Virtual address: three parts

- Level-one page number (10 bits)
- Level-two page number (10 bits)
- Offset (12 bits)

PTE in 1st level page table contains physical frame # for one 2nd level page table

2nd level page table has actual physical frame numbers for the memory address

Page Table Size for 32bit System

Modern Systems/Applications

- 32 bits virtual address
- System with 1GB physical memory \( \rightarrow \) 30 bits physical address
- Suppose the size of one pageframe is 4KB (12 bits)

Page table size

- # of virtual pages: \( 32 \times 2^{20} = 2^{32} \) PTEs
- Page table size = PTE size \( \times 2^{10} = 4 \) MB per process \( \rightarrow 2^{10} \) frames

If there are 128 processes

- Page tables occupy 128 \( \times 4MB = 512 \) MB
- 50% of memory will be used by page tables?

Two-Level Page Tables of Linux

Why it is good?

- We don't have to allocate all levels initially
- They don't have to be continuous

Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes

- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space

How can we get smaller page table?!
Example: 2-level Address Translation

Memory Requirement of Page Tables

- Only the 1st level page table and the required 2nd level page tables need to be in memory.

Example: a process with working-set size of 32 MB (recall that 1GB memory and 32 bits virtual address)
- Assume the page/frame size is 4KB, the page table are two-levels page table as the last slide: 10bit for the first level, 10bit for the 2nd level.
- Question: what is the minimum memory should be used for page table

Memory Requirement of Page Tables

- 32MB/4KB → 8K virtual pages (minimum) to load this process.
- Assuming each table entry is 4 Bytes, then one page can store 1K page table entries. So we need at least 8K/1K = 8 pages, which will be used as second level pages.
- We need a first level page table, for which we will allocate one page.
- So overall, we need 9 pages (9 X 4KB = 36 KB) to maintain the page tables.

Page table size

- 32bit machine, page size 4k, each entry 4 bytes, one level page table (full 4GB linear address)
  \[ \text{Page table size} = 2^{20} \times 4 \text{bytes} = 2^{22} = 4 \text{M} \]
- 32bit machine, page size 4k, each entry 4 bytes, two level page table (two pages: 0x00000000, and 0xFFFFF000)
  \[ \text{Page table size} = (2^{10} \times 0 \text{entries}) \times 4 \text{ bytes} + (2^{10} \times 1 \text{ entries} \times 4 \text{ bytes}) \times 2 = 12 \text{ Kbytes} \]

Quiz

- Why the page table has to be physically continuous?
- If we have accessed two pages (0x00000000 and 0x00201000), what is the size of page table?

Hashed Page Tables

- The virtual page number is hashed into a page table that contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.
- Clustered page tables, each entry refers to several pages…
Inverted Page Table

Page table:
- Translate virtual addresses to physical addresses
- We will need this for every memory reference

Drawback?
- Each page table has millions of entries
- If we touch two pages, three physical pages may need for table

Why we need inverted page table?
- Solve this problem

Why we need inverted page table?
- Solve this problem

Inverted Page Table

- One entry for each real page (frame)
  of memory
- Entry consists of the virtual address
  of the page stored in that real
  memory location, with information
  about the process that owns that
  page
- Decreases memory needed to store
  each page table, but increases time
  needed to search the table when a
  page reference occurs. Also hard to
  implement shared memory
- Use hash table to limit the search to
  one — or at most a few — page-
  table entries

Inverted Page Table (cont.)

Translation Lookaside Buffer (TLB)

Small Hardware: fast
- Store recent accessed mapping of
  page \( \rightarrow \) frame (64 ~ 1024 entries)
- If desired logical page number is
  found, get frame number from TLB
- If not,
  - Get frame number from page table in
    memory
  - Use standard cache techniques
  - Replace an entry in the TLB with the
    logical & physical page numbers from
    this reference
  - Contains complete page table entries
    for small number of pages

Example TLB

Address Translation with TLB

Integrating VM and Cache

Most caches "physically addressed"
- Accessed by physical addresses
- Allows multiple processes to have blocks in cache at same
  time else context switch \( \Rightarrow \) cache flush
- Allows multiple processes to share pages
- Cache doesn’t need to be concerned with protection issues
  - Access rights checked as part of address translation
- Perform address translation before cache lookup
  - Could involve memory access itself (to get PTE)
  - So page table entries can also be cached

What happens when cpu performs a context switch?
Integrating TLB and Cache

"Translation Lookaside Buffer" (TLB)

Understanding the workflow!!!

The procedure of memory access

① Uses special associative cache (TLB) to speed up translation
② Turns to full page translation tables in memory, if TLB misses. This is much slower.
③ Falls back to OS if page translation table misses
  ➢ Such a reference to an unmapped address causes a page fault
④ If the memory address is fetched, check whether the block in cache or not.
⑤ If yes, getting the data from the cache.
⑥ Otherwise, fetch the cache line from the memory into the cache.

Memory Accesses Time

Assuming:
➤ TLB lookup time = a
➤ Memory access time = m
Hit ratio (h) is percentage of time that a logical page number is found in the TLB
➤ More TLB entries usually means higher h
Effective Access Time (EAT) is calculated (don’t include cache effect)
➤ EAT = (m + a)h + (m + m + a)(1-h) = a + (2-h)m
Interpretation
➤ Reference always requires TLB lookup, 1 memory access
TLB misses also require an additional memory reference

Effective Access Time: An Example

TLB Lookup = 20 nanoseconds
Memory cycle time is 100 nanoseconds
Hit ratio = 80%
Effective Access Time (EAT) would be
EAT = (100 + 20) 0.8 + (200 + 20)(1 – 0.8)
= 140 nanosecond
40% slow down
What about 98% hit rate? What the EAT is?

Memory Management: outline

Virtual Memory
Demand Paging
Physical Memory Management (Buddy and Slap)
Page Replacement
Memory Mapping (mmap)
Connecting with User Space Memory

Virtual Memory

Basic idea: allow OS to allocate more memory than the real
Program uses virtual addresses
➤ Addresses local to the process
➤ Can be any size → limited by # of bits in address (32/64)
➤ 32 bits: 4G
Virtual memory >> physical memory
Motivations for Virtual Memory

- Virtual pages can exceed physical memory size
- Simplify memory management: an extremely large, uniform logic memory
  - Each process has its own address space, free the concern of memory-storage
  - Only “active” code and data is actually in memory
- Provide protection
  - One process can’t interfere with another
  - Because they operate in different address spaces
  - User process cannot access privileged information
  - Different sections of address spaces have different permissions

Virtual address space

- The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory.

How to bring the program to the memory?

- One method:
  - Loading the whole program at startup
  - Problems:
    - We may not need the program initially
    - Slow starting a program
    - Most of loading can be useless, most of a program may not be utilized in the execution

  Load page when they are needed

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated.
  - v ⇒ in-memory
  - i ⇒ not-in-memory
- Initially bit is set to i on all entries
- During address translation, if valid–invalid bit in page table entry is i ⇒ page fault (trap)

Page Fault

1. Reference to a page.
   If invalid reference ⇒ abort
2. If not in memory, page fault occurs (trap to OS)
3. OS allocates an empty frame
4. Swap page into frame
5. Reset page tables, set validation bit = v
6. Restart the instruction that caused the page fault
Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
  - If $p = 0$ no page faults
  - If $p = 1$, every reference is a fault

- Effective Access Time (EAT)
  \[ EAT = (1 - p) \times \text{memory_access} + p \times \text{page_fault_time} \]
  - \text{page_fault_time} depends on several factors
  - Save user reg and proc state, check page ref, read from the disk there might be a queue, (CPU can be given to another proc), get interrupt, save other user reg and proc state, correct the page table, put this process into ready queue… Due to queues, the \text{page_fault_time} is a random variable

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
  \[ EAT = (1 - p) \times 200 + p \times 8,000,000 \]
  \[ = 200 + p \times 7,999,800 \]
- If one out of 1,000 access causes a page fault, then
  \[ EAT = 8.2 \text{ microseconds} \]
  This is a slowdown by a factor of 40!!
- If we want just 10% performance degradation, then $p$ should be $220 > (1 - p) \times 200 + p \times 8 \text{ milliseconds}$
  \[ p < 0.0000025 \text{, i.e., } 1 \text{ page fault out of 400,000 accesses} \]

Copy-On-Write

- Fork procedure:
  - Creating a copy of the parent’s address space for the child
  - However, copying everything will be unnecessary if child processes invoke the \text{exec()} system call immediately

- Copy-on-write:
  - Parent and child are sharing the same pages. But if one process is writing on a shared page, a separate page will be created

Buddy System (Dividing)

- Two continuous blocks with the same size the first one will start as $2^n$

Free Page’s List
Page Allocation

Example: Need to allocate 65 contiguous page frames.
- Look in list of free 128-page-frame blocks.
- If free block exists, allocate it, else look in next highest order list (here, 256-page-frame blocks).
- If first free block is in 256-page-frame list, allocate a 128-page-frame block and put remaining 128-page-frame block in lower order list.
- If first free block is in 512-page-frame list, allocate a 128-page-frame block and split remaining 384 page frames into 2 blocks of 256 and 128 page frames. These blocks are allocated to the corresponding free lists.

Question: What is the worst-case internal fragmentation?

Buddy De-Allocation

- When blocks of page frames are released the kernel tries to merge pairs of "buddy" blocks of size $b$ into blocks of size $2b$.
- Two blocks are buddies if:
  - They have equal size $b$.
  - They are located at contiguous physical addresses.
  - The address of the first page frame in the first block is aligned on a multiple of $2b^2$.
- The process repeats by attempting to merge buddies of size $2b$, $4b$, $8b$ etc…

Slab Allocator

- Slab is one or more physically contiguous pages
- Cache consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache consists of objects — instantiations of the data structure
- When cache created, filled with objects marked as free
- When structures stored, objects marked as used
- If slab is full, next object is allocated from empty slab. If no empty slabs, new slab allocated
  - Benefits include:
    - no fragmentation,
    - memory request is satisfied quickly

Slab Allocator

- Performs the following functions
  - Allocate memory
  - Initialize objects/structures
  - Use objects/structures
  - Deconstruct objects/structures
  - Free memory
- /proc/slabinfo – gives full information about memory usage on the slab level. (see also /usr/bin/slabtop)
Page Replacement

- No free frame in memory, a page needs to be replaced.
- Pages that are replaced might be needed again later.
- We need algorithms to minimize the number of page faults.
- Include other improvement, e.g., use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk

Basic Page Replacement

- Find the desired page on disk
- If there is a free frame, use it
- If there is no free frame, use a page replacement algorithm
  1. Select a victim frame, swap it out (use dirty bit to swap out only modified frames)
  2. Bring the desired page into the (newly) free frame;
  3. Update the page and frame tables
- Restart the process

Page Replacement Algorithms

- How to select the victim frame?
  - You can select any frame, the page replacement will work; but the performance???
  - Gives the lowest page-fault rate
- Evaluate an algorithm by running it on a particular string of memory references (reference string) and compute the number of page faults on that string
- In all our examples, we will have 3 frames and the following reference string

\[
\begin{array}{cccccccccccc}
7 & 0 & 1 & 2 & 0 & 3 & 0 & 4 & 2 & 9 & 0 & 3 & 2 & 1 & 2 & 0 & 1 & 7 & 0 & 1
\end{array}
\]

First-In-First-Out (FIFO) Algorithm

- Maintain an FIFO buffer
  - The page used before may not be needed
  - An array used early, might be used again and again
- Easy to implement
- Belady’s Anomaly: more frames ⇒ more page faults

FIFO Illustrating Belady’s Anomaly

Optimal Algorithm

- Replace a page that will not be used for longest time
- How do you know the future?
- Used for measuring how well your algorithm performs
Least Recently Used (LRU) Algorithm

- Use recent past as an approximation of the future
- Select the page that is not used for a long time...
  
  - OPT if you look at it from backward
  - NO Belady's Anomaly: so more frames ⇒ less page faults

Given the reference string of page accesses: 1 2 3 4 1 2 1 3 1 4 and a system with three page frames, what is the final configuration of the three frames after the true LRU algorithm is applied?

Problem of LRU:
How to implement it efficiently?
Full LRU needs to sort all time of reference.

LRU Algorithm (Cont.)

- Counter (logical clock) implementation
  
  - Increase the counter every time a page is referenced
  - Save it into time-of-use field
  - Find one with the smallest time-of-use value
  - Problems: Counter overflow and linear search perfor

- Stack implementation – keep a stack of page numbers in a double link form:
  
  - Page referenced: move it to the top
  - requires 6 pointer ops to be changed
  - No search for replacement
  - Least recently used one is at the bottom

LRU Approximation Algorithms

- Reference bit
  
  - With each page associate a reference bit, initially = 0
  
  - When page is referenced, set this bit to 1 by hardware
  
  - Replace the one which is 0 (if one exists)
  
  - We do not know the order, however
  
  - Additional bits can help to gain more ordering information

- Second chance Alg (uses one reference bit)
  
  - FIFO with an inspection of ref bit
  
  - If ref bit is 0, replace that page
  
  - set ref bit to 1

- If ref bit is 1, give a second chance
  
  - set ref bit to 0
  
  - leave page in memory
  
  - Arrival time set to current time
  
  - go to next one
  
  - Additional bits can help to avoid replacing modified pages

User-Level Memory Mapping

```c
void *mmap(void *start, int len, int prot, int flags, int fd, int offset)
```

- Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start`
  
  - `start`: may be 0 for "pick an address"
  
  - `prot`: PROT_READ, PROT_WRITE, ... 
  
  - `flags`: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ... 

- Return a pointer to start of mapped area (may not be `start`)

Two different types of mmaped region

- Anonymous
  
  - No backup on files

- File-backed mapping
  
  - Backed up by a file.
User-Level Memory Mapping

void *mmap(void *start, int len, int prot, int flags, int fd, int offset)

- start (or address chosen by kernel)
- len bytes
- offset (bytes)
- Disk file specified by file descriptor fd
- Process virtual memory

Threads: Outline

- Motivation and thread basics
- Resources requirements: thread vs. process
- Thread implementations
  - User threads: e.g., Pthreads and Java threads
  - Kernel threads: e.g., Linux tasks
  - Map user- and kernel-level threads
  - Lightweight process and scheduler activation
- Other issues with threads: process creation and signals etc.
- Threaded programs
  - Thread pool
  - Performance vs. number of threads vs. CPUs and I/Os

Context Switches of Processes: Expensive

- Context switch between processes
  - Save processor registers for current process
  - Load the new process’s registers
- Switch address spaces – expensive
  - Hardware cache
  - Memory pages (e.g., TLB content)

Thread vs. Process

- Responsiveness
  - Part of blocked
- Resource Sharing
  - Memory, open files, etc.
- Economy
  - Creation and switches
- Scalability
  - Increase parallelism

Process: Alternative View

- Process = thread + code, data, and kernel context

Process with Two Threads

- Thread 1
  - Program context:
    - Data registers
    - Condition codes
    - Stack pointer (SP)
    - Program counter (PC)
  - Stack
  - Kernel context:
    - VM structures
    - Descriptor table
    - brk pointer

- Thread 2
  - Program context:
    - Data registers
    - Condition codes
    - Stack pointer (SP)
    - Program counter (PC)
  - Stack
  - Kernel context:
    - VM structures
    - Descriptor table
    - brk pointer
Resources for Threads

- **Shared** resources among threads (per process items)
  - Address space (e.g., codes)
  - Global variables/data
  - Open Files and other resources etc.

- **Separated** resources for each thread
  - Machine state: registers (e.g., PC)
  - Running stacks
  - Private data

Threads vs. Processes

- Threads and processes: similarities
  - Each has its own logical control flow
  - Each can run concurrently with others
  - Each is context switched (scheduled) by the kernel

- Threads and processes: differences
  - Threads share code and data, processes (typically) do not
  - Threads are less expensive than processes
    - Process control (creation and exit) is more expensive than thread control
    - Context switches: processes are more expensive than for threads

Pros and Cons of Thread-Based Designs

- **Pros**
  - Easy to share data structures between threads
    - e.g., logging information, file cache
  - Threads are more efficient than processes

- **Cons**
  - Unintentional sharing can introduce subtle and hard-to-reproduce errors!

Multithreading Models: Many-to-One

- Many user-level threads mapped to a single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

Multithreading Models: One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later
**One-to-one Model**

- **Pros:**
  - Scalable parallelism (concurrency)
  - Thread will not block a whole process

- **Cons:**
  - Expensive synchronization (system call is required if a lock can’t be acquired)
  - Expensive creation (3.5 slower)
  - Kernel resource, e.g. stack and kernel structure

**Multithreading Models: Many-to-Many Model**

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

**Many-to-Many Model**

- **Pros:**
  - Cheap Resource, not all user threads should create a kernel thread
  - Synchronization mainly at user-level
  - Context switch may not involve system calls

- **Cons:**
  - Difficult cooperation between kernel scheduler and user scheduler
  - How to decide the number of kernel threads?

**Pthreads: POSIX Thread**

- **POSIX**
  - Portable Operating System Interface (for Unix)
  - Standardized programming interface

- **Pthreads**
  - Thread implementations adhering to POSIX standard
  - API specifies behavior of the thread library: defined as a set of C types and procedure calls
  - Common in UNIX OS (Solaris, Linux, Mac OS X)
  - Support for thread creation and synchronization

**Linux Threads**

- Linux uses the term *task* (rather than process or thread) when referring to a flow of control

- Linux provides clone() system call to create threads
  - A set of flags, passed as arguments to the clone() system call determine how much sharing is involved (e.g. open files, memory space, etc.)

- Linux: 1-to-1 thread mapping
  - NPTL (Native POSIX Thread Library)

---

**Pthreads: Thread/Synchronization APIs**

<table>
<thead>
<tr>
<th>Thread Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_create</td>
<td>Create a new thread in the caller’s address space</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>Terminate the calling thread</td>
</tr>
<tr>
<td>pthread_join</td>
<td>Wait for a thread to terminate</td>
</tr>
<tr>
<td>pthread_mutex_init</td>
<td>Create a new mutex</td>
</tr>
<tr>
<td>pthread_mutex_destroy</td>
<td>Destroy a mutex</td>
</tr>
<tr>
<td>pthread_mutex_lock</td>
<td>Lock a mutex</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>Unlock a mutex</td>
</tr>
<tr>
<td>pthread_cond_init</td>
<td>Create a condition variable</td>
</tr>
<tr>
<td>pthread_cond_destroy</td>
<td>Destroy a condition variable</td>
</tr>
<tr>
<td>pthread_cond_wait</td>
<td>Wait on a condition variable</td>
</tr>
<tr>
<td>pthread_cond_signal</td>
<td>Release one thread waiting on a condition variable</td>
</tr>
</tbody>
</table>

---

**Linux Threads**

- Linux uses the term *task* (rather than process or thread) when referring to a flow of control

- Linux provides clone() system call to create threads
  - A set of flags, passed as arguments to the clone() system call determine how much sharing is involved (e.g. open files, memory space, etc.)

- Linux: 1-to-1 thread mapping
  - NPTL (Native POSIX Thread Library)
Other Issues: Process Creation in Thread

- What will happen if one thread in a process call fork() to create a new process?
  - How many threads in the new process?
- Duplicate only the invoking thread
  - exec(): will load another program
  - Everything will be replaced anyway
- Duplicate all threads
  - What about threads performed blocking system call?

Shared Variables in Threaded C Programs

- Question: Which variables in a threaded C program are shared?
  - The answer is not as simple as “global variables are shared” and “stack variables are private”
- Requires answers to the following questions:
  - What is the memory model for threads?
  - How are variables mapped to memory?
  - How many threads might reference each variable?

A variable \( x \) is shared if and only if multiple threads reference some instance of \( x \)

Threads Memory Model

- Conceptual model:
  - Multiple threads run in the same context of a process
  - Each thread has its own separate thread context
    - Thread ID, stack pointer, PC, and GP registers
  - All threads share the remaining process context
    - Code, data, heap, and shared library segments
    - Open files and installed handlers
- Operationally, this model is not strictly enforced:
  - Register values are truly separate and protected, but...
  - Any thread can read and write the stack of any other thread

Example Program to Illustrate Sharing

```
char **ptr; /* global */
int main()
{
  int i;
  pthread_t tid;
  char *msgs[2] = {
    "Hello from foo",
    "Hello from bar"
  };
  ptr = msgs;
  for (i = 0; i < 2; i++)
    pthread_create(&tid, NULL, thread, (void *)i);
  pthread_exit(NULL);
}

/* thread routine */
void *thread(void *vargp)
{
  int myid = (int)vargp;
  static int cnt = 0;
  printf("[%d]: %s (svar=%d)\n", myid, ptr[myid], ++cnt);
}
```

Peer threads reference main thread’s stack indirectly through global ptr variable

Mapping Variable Instances to Memory

- Global variables
  - Def: Variable declared outside of a function
  - Virtual memory contains exactly one instance of any global variable
- Local variables
  - Def: Variable declared inside function without static attribute
  - Each thread stack contains one instance of each local variable
- Local static variables
  - Def: Variable declared inside function with the static attribute
  - Virtual memory contains exactly one instance of any local static variable.
Thread Pool

- Pool of threads
  - Threads in a pool where they wait for work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
- Adjust thread number in pool
  - According to usage pattern and system load

Thread Pool Implementation

- Work queue
  - Fixed number of threads
- Other possible problems
  - Deadlock
  - Resource thrashing
  - Thread leakage
  - Overload

Performance of Threaded Programs

- Suppose that the processing of each request
  - Takes X seconds for computation; and
  - Takes Y seconds for reading data from I/O disk
- For single-thread program/process
  - A single CPU & single disk system
  - What is the maximum throughput (i.e., the number of requests can be processed per second)?
  - Example: suppose that each request takes
    - 2ms for computation
    - 8ms to read data from disk
    - $1000/10\text{ms} = 100$

Performance of Threaded Programs (cont)

- Multi-thread performance improvement
  - Single CPU & single disk system
  - How many threads should be used?
  - What is the maximum throughput (i.e., the number of requests can be processed per second)?
  - Example: suppose that each request takes
    - 2ms for computation
    - 8ms to read data from disk
    - Assuming that we have 8 cores and 1 disk
    - $1000/8\text{ms} = 125$

Performance of Threaded Programs (cont)

- What about m-CPU and n-disk system
  - Maximum throughput and # of threads? (X: computation, Y: IO for each task)
  - Throughput \( \rightarrow \frac{1}{\max(X/m, Y/n)} \)
  - if \( X/m < Y/n \), \# = n + m'
    where \( m' = \min(k | X/k \leq Y/n, 1 \leq k \leq m) \);
  - Similarly, if \( X/m > Y/n \), \# = m + n'
    where \( n' = \min(k | X/m > Y/k, 1 \leq k \leq n) \);
- Other issues:
  - When I/O disk is bottleneck, adding more CPUs will NOT help improve the throughput
  - What about heterogeneous disks and CPUs?!
**Concurrent Access to Shared Data**

- Two threads A and B have access to a shared variable “Balance”

  **Thread A:**
  
  Balance = Balance + 100

  A1. LOAD R1, BALANCE  
  A2. ADD R1, 100  
  A3. STORE BALANCE, R1

  **Thread B:**
  
  Balance = Balance - 200

  B1. LOAD R1, BALANCE  
  B2. SUB R1, 200  
  B3. STORE BALANCE, R1

---

**What is the problem then?**

- **Observe:** In a time-shared system, the exact instruction execution order cannot be predicted

**Scenario 1:**

A1. LOAD R1, BALANCE  
A2. ADD R1, 100  
A3. STORE BALANCE, R1  
Context Switch!  
B1. LOAD R1, BALANCE  
B2. SUB R1, 200  
B3. STORE BALANCE, R1

- **Sequential correct execution**
  - Balance is effectively decreased by 100!

**Scenario 2:**

B1. LOAD R1, BALANCE  
B2. SUB R1, 200  
Context Switch!  
A1. LOAD R1, BALANCE  
A2. ADD R1, 100  
A3. STORE BALANCE, R1  
Context Switch!  
B3. STORE BALANCE, R1

- **Mixed wrong execution**
  - Balance is effectively decreased by 200!!!

---

**Race Conditions**

- Multiple processes/threads write/read shared data and the outcome depends on the particular order to access shared data are called race conditions
  - A serious problem for concurrent system using shared variables!

**How do we solve the problem?!**

- Need to make sure that some high-level code sections are executed atomically
  - Atomic operation means that it completes in its entirety without worrying about interruption by any other potentially conflict-causing process

---

**What is Synchronization?**

- Cooperating processes/threads share data & have effects on each other → executions are NOT reproducible with non-deterministic exec. speed
  - Concurrent executions
    - Single processor → achieved by time slicing
    - Parallel/distributed systems → truly simultaneous
  - Synchronization → getting processes/threads to work together in a coordinated manner

---

**Mutual Exclusion**

- Critical section (CS)
  - A section of code that modify the same shared variables/data that must be executed mutually exclusively in time
  - General structure for processes/threads with CS
    - **entry section:** The code which requests permission to enter the critical section.
    - **critical section:** as above
    - **exit section:** The code which removes the mutual exclusion.
    - **remainder section:** Everything else
General Structure for Critical Sections

```
do {
    entry section
    critical section
    exit section
    remainder section
} while (1);
```

In the entry section, the process requests "permission".

Requirements for CS Solutions

- **Mutual Exclusion**
  - At most one process/thread in its CS at any time
- **Progress**
  - If all other processes/threads are in their remainder sections, a process/thread is allowed to enter its CS
  - Only those processes that are not in their remainder section can decide which process can enter its CS next, and this decision cannot be postponed indefinitely.
- **Bounded Waiting**
  - Once a process has made a request to enter its CS, other processes can only enter their CSs with a bounded number of times.

A Simple Solution

- `int turn;`
  - indicate whose turn to enter CS
- `T0 and T1: alternate between CS and remainder`

```
Process 0:
----------
while(TRUE) {
    while (turn != 0): critical section
    turn = 1; remainder section
}
```

Process 1:
----------
```
while(TRUE) {
    while (turn != 1): critical section
    turn = 0; remainder section
}
```

What is the problem with this solution?
- Mutual exclusion?
- Progress?
- Bounded Waiting?

Peterson's Solution

- Three shared variables: `turn` and `flag[2]`

```
Process 0 loop:
---------------
flag[0] = 1;
while (flag[0] == 1) & (turn==1):
    critical section
flag[0] = 0;
remainder section
```

Process 1 loop:
---------------
```
flag[1] = 1;
while (flag[1] == 1) & (turn==0):
    critical section
flag[1] = 0;
remainder section
```

Hardware Support for Synchronization

- **Synchronization**
  - Need to test and set a value atomically
- **IA32 hardware provides a special instruction: xchg**
  - When the instruction accesses memory, the bus is locked during its execution and no other process/thread can access memory until it completes!!!
  - Other variations: xchgb, xchgw, xchg1
- **Other hardware instructions**
  - TestAndSet (a)
  - Swap (a,b)

Hardware Instruction **TestAndSet**

- **The TestAndSet instruction tests and modifies the content of a word atomically (non-interruptable)**
- **Keep setting the lock to 1 and return old value.**

```
bool TestAndSet(bool *target) {
    boolean m = *target;
    *target = true;
    return m;
}
```

**What's the problem?**
1. Busy-waiting, waste cpu
2. Hardware dependent, not bounded-waiting
Another Hardware Instruction: Swap

Swap contents of two memory words

```c
void Swap (bool *a, bool *b){
    bool temp = *a;
    *a = *b;
    *b = temp;
}
```

What’s the problem?
1. Busy-waiting, waste cpu
2. Hardware dependent, not bounded-waiting

Semaphores

Synchronization without busy waiting
- Motivation: Avoid busy waiting by blocking a process execution until some condition is satisfied

Semaphore S – integer variable
- Two indivisible (atomic) operations:
  - wait(s) (also called P(s) or down(s) or acquire());
  - signal(s) (also called V(s) or up(s) or release());
- User-visible operations on a semaphore
- Easy to generalize, and less complicated for application programmers

Semaphore Operations
- Semaphore is an integer.
- wait(s): //wait until s.value > 0;
- signal(s): s.value++; /* Executed atomically! */
- A process execute the wait operation on a semaphore with value <=0 is blocked
- Blocked → non-runnable state, yield CPU to others
- If s=1, wake up only one blocked process; which one?

Implementation is a spinlock: waste CPU, but no need for context switch

Semaphore Usage
- Counting semaphore – Can be used to control access to a given resources with finite number of instances
- Binary semaphore – integer value can range only between 0 and 1; Also known as mutex locks

Attacking CS Problem with Semaphores
- Shared data
  - semaphore mutex = 1; /* initially mutex = 1 */
- For any process/thread
  ```c
do {
    ...
    wait(mutex);
    critical section
    signal(mutex);  /* remainder section */
} while(1);
```
Revisit “Balance Update Problem”

- Shared data:
  int Balance;
  semaphore mutex; // initially mutex = 1

- Process A:
  wait (mutex);
  Balance = Balance – 100;
  signal (mutex);

- Process B:
  wait (mutex);
  Balance = Balance – 200;
  signal (mutex);

Semaphore for General Synchronization

- Execute code B in P₂ after code A is executed in P₁.
- Use semaphore flag: what is the initial value?
- Code
  
  \[
  \begin{align*}
  &P_1: \\
  &P_2: \\
  &\text{signal}(\text{flag})
  \end{align*}
  \]

What about 2 threads wait for 1 thread? Or 1 thread waits for 2 threads?

Producer-Consumer Problem

With Bounded-Buffer

- Need to make sure that
  - The producer and the consumer do not access the buffer area and related variables at the same time
  - No item is available to the consumer if all the buffer slots are empty.
  - No slot in the buffer is available to the producer if all the buffer slots are full

Producer-Consumer Codes

- Producer
  ```
  do {
    produce an item in nextp
    while (counter == n) {
      buffer[in] = nextp;
      in = (in+1) % n;
      counter++;
      wait(mutex);
    }
  } while (1)
  ```

- Consumer
  ```
  do {
    remove an item from buffer to nextc
    wait(mutex);
    signal(mutex);
    wait(full);
    add nextc to buffer
    while (1)
  } while (1)
  ```

What Semaphores are needed?

- semaphore mutex, full, empty;

What are the initial values?

- Initially:
  - full = 0 /* The number of full buffer slots */
  - empty = n /* The number of empty buffer slots */
  - mutex = 1 /* controlling mutual access to the buffer pool */

Producer/Consumer Loops

- Producer Loop
  ```
  produce an item in nextp
  while (counter == n) {
    buffer[in] = nextp;
    in = (in+1) % n;
    counter++;
    wait(mutex);
  }
  ```

- Consumer Loop
  ```
  while (counter == 0) {
    nextp = buffer[out];
    out = (out+1) % n;
    counter--;
    consume the item in nextp;
  }
  ```