Overview

Summary 8: p441 Summary 9: p490 Summary 10: p552

Chapter 8 Review

Deadlock Conditions (Slide 8.10)

- **Mutual exclusion**;
- **Hold and wait**;
- **No preemption**
- **Circular wait**.

Resource Allocation Graph (Slide 8.11)

- Two types of vertex: processes, resources;
- Two types of edge: **request edge** ($P_i \rightarrow R_j$) & **assignment edge** ($R_j \rightarrow P_i$).

Cycle and Deadlock (Slide 8.15)

- If graph contains no cycle, no deadlock;
- If graph contains a cycle, and only one instance per resources type, then deadlock;
- If graph contains a cycle, and several instances per resource type, the possibly deadlock.

Method for handling deadlocks (Slide 8.16)

- Ensure that the system will never enter a deadlock state;
  - Deadlock prevention;
  - Deadlock avoidance;
- Allow the system to enter a deadlock state and then recover;
- Ignore the problem and pretend that deadlocks never occur in the system.

Deadlock prevention (Slide 8.17~8.19): Invalidate one of four necessary conditions for deadlock.

- **Mutual Exclusion**: sharable resources, such as read-only files;
- **Hold and Wait**: require process to request and be allocated all its resources before it begins execution, or allow processes to request resources only when the process has none allocated to it. But it may cause *low resources utilization* and *possibly starvation*.
- **No preemption**: If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released; Preempted resources are added to the list of resources for which the process is waiting; and process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- **Circular Wait**: impose a total ordering of all resource types, and require that each process requests resources in *an increasing order of enumeration*. *Resources must be acquired in order*.

Deadlock Avoidance (Slide 8.20): requires that the system has some additional **a priori** (which means from the earlier) information available, to **make sure that the system must not enter an unsafe state**.

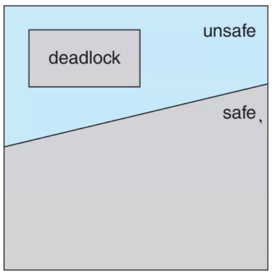
- Simplest and most useful model requires that each process *declare the maximum number of resources of each type that it may need*.
- **The deadlock avoidance algorithm** dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition;

- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demand of the processes.

Safe State (Slide 8.21) If there exists a process execution sequence that the process can follow the sequence to execute, and finish successfully, that is, no deadlock.

Unsafe State: the contrary to safe state.

Safe/Unsafe and Deadlock (Slide 8.22-23)



- In safe state, no deadlock;
- In unsafe state, possible deadlock.

Avoidance Algorithms (Slide 8.24)

- Single instance of a resource type: use a resource-allocation graph;
- Multiple instances of a resource type: use the Banker's Algorithm.

Resource-Allocation Graph Scheme (Slide 8.25)

- **Claim Edge** $P_i \to R_j$ indicated that process P_i may request resource R_j; represented by a *dashed* line.
- Claim edge convert to request edge when a process request a resource;
- Request edge convert to assignment edge when the resource is allocated to the process;
- When the resource is released by the process, assignments reconverts to a claim edge;
- Resource must be claimed *a priori* (which means from the earlier) in the system.
- **Methods** (Slide 8.28): suppose that process P_i request a resource R_j, then the resource can be granted only if converting the request edge to an assignment edge does not form a cycle in the allocation graph.

Banker's Algorithm (Slide 8.29) Multiple instances of resources, and each process must a priori claim a maximum use; when a process requests a resource, it may have to wait; when the process gets all the resources it needs, it must return in a finite amount of time.

- **Available** vector;
- **Max** matrix;
- **Allocation** matrix;
- **Need** matrix = **Max - Allocation**

Process example: Slide 8.31 ~ Slide 8.35

Deadlock detection: allows system to enter deadlock state, and use detection algorithm to detect deadlocks, and use recovery scheme to go back to the safe state.

- **Wait-for graph**: nodes are processes, $P_i \to P_j$ if P_i is waiting for P_j.

- If single instance of a resource type, then periodically invoke an algorithm that searches for a cycle in the wait-for graph. If there exists a cycle, there exists a deadlock. $O(n^2)$ times to detect.

- **Detection Algorithm** (Slide 8.40~8.43): multiple instances of a resource type, then use the same method as the Banker's Algorithm to detect deadlock.

- **Usage** (Slide 8.44)
    - **Roll back**: for every disjoint cycle, we need to roll back one process.
    - If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.
- **Recovery from deadlock**:
    - **Process termination**: abort one process at a time until the deadlock is eliminated (abort the process in circular wait). And there are many orders that you can choose to terminate the processes (Slide 8.45).
    - **Resource Preemption**: selecting a victim process, **roll back** to some safe state, restart process from that state. It may cause *starvation* because of choosing the same victim every time.

Chapter 9 Review

Memory Protection (Slide 9.7): need to censure that a process can only access those addresses in its address space, using a pair of ***base register*** (also called ***relocation register***) and ***limit register*** to define the logical address space of a process.

- **Hardware address protection** (Slide 9.8): add some checking unit in the circuit to ensure the security.

Address Binding (Slide 9.9): maps the virtual address space to the physical address space. It could happen at three different stages (Slide 9.10)

- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile if starting location changes;
- **Load time**: must generate *relocatable code* if memory location is not known at compile time;
- **Execution Time**: binding delayed until run time if the process can be moved during its execution from one memory segment to another (need hardware support for address maps).

Address Spaces (Slide 9.12)

- **Logical Address** is generated by CPU, and it also referred to as **virtual address**. **Logical address space** is the set of all logical addresses generated by a program.
- **Physical address** is address seen by the memory unit. **Physical address space** is the set of all physical addresses generated by a program.

Memory Management Unit (MMU) (Slide 9.13): the unit to perform **address binding**, maps logical address to physical address.

Simple scheme (Slide 9.14~9.15): base-register scheme.

Dynamic Loading (Slide 9.16): routine is not loaded until it is called. It can lead to better memory-space utilization because unused routine is never called. All routines kept on disk in relocatable load format. *No special support from OS is required, except providing some libraries for implementation*.

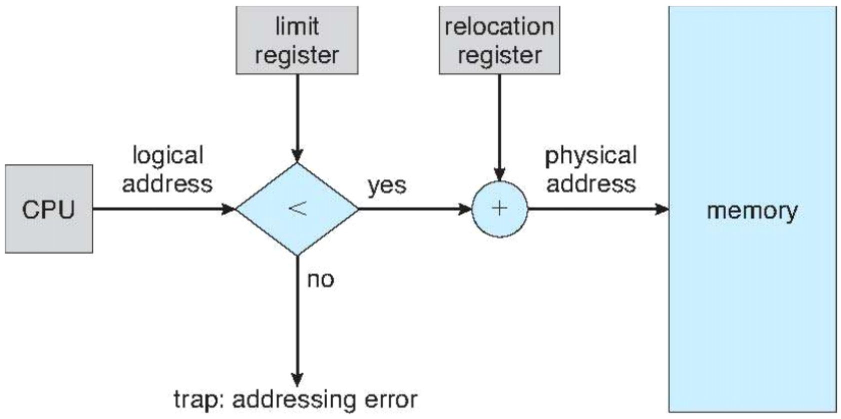
Static Linking: system libraries and program code combined by the loader into the binary program image.

Dynamic Linking (Slide 9.17): linking postponed until execution time.

- Some small piece of code, **stub**（桩代码）, used to locate the appropriate memory resident library routine. Stub will replace itself with the address of the routine and execute the

routine;
- OS checks if routine is in processes' memory address;
- Dynamic linking is useful for libraries, also called **shared libraries**.

Main memory (Slide 9.18)



- Resident OS, usually held in low memory with interrupt vector;
- User processes then held in high memory;
- Each process contained in **single, contiguous section of memory**;
- Relocation registers (base register) used to protect user processes from each other, and from changing OS code and data;
- MMU maps logical address *dynamically*;
- Allow actions such as kernel code being *transient* and kernel changing size.

Variable partition（可变分区） (Slide 9.21)

- **Hole**: block of available memory; holes of various size are scattered throughout memory;
- OS maintains information about both allocations partitions and free partitions (holes);
- *Storage-allocation problem* (Slide 9.22): 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 (smallest leftover hole, need to search the whole list);
    - **Worst-fit**: allocate the biggest hole (biggest leftover hole, need to search the whole list).

Fragmentation (Slide 9.23)

- **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; this size difference is memory internal to a partition, but not being used.
- **50-percent rule**: given n blocks allocated, $0.5 \cdot n$ blocks lost to fragmentation (unusable).
- **Compaction**: reduce external fragmentation.
    - Shuffle memory contents to place all free memory together in one large block;
    - Possible only if relocation is dynamic, and is done at execution time.

Paging (Slide 9.25)

- **Frame**: physical memory into fixed-sized blocks (also backing store);
- **Pages**: logical memory into blocks of same size;
- **Page table** (Slide 9.32): translate logical to physical addresses.

    - **PTBR** points to the page table (page table base register);
    - **PTLR** indicates the size of page table (page table length register);
    - **Translation look-aside buffers (TLBs)**: also called *associative memory*, acts as the cache of the page table.
        - ***Address-space identifiers (ASIDs)*** (Slide 9.33) in every TLB entry indicates the process the entry comes from.
- Address generated by CPU is divided into: page number p and page offset d;
    - page number is used as an index into a page table to find out the corresponding *base address* of the page in physical memory;
    - page offset is combined with base address to define the physical memory address that is sent to the memory unit.
- **Internal fragmentation calculation**
    - Worst: frame size - 1 byte;
    - On average: 1/2 frame size.
- **Effective Access Time (EAT)** calculation.
- **Memory protection** in page table (Slide 9.37): **valid-invalid bits** attached to each entry in page table. (Another way is to use *PTLR* to restrict length of page table.)
    - Valid indicates the associated page is in the process' logical address space, and is thus a legal page;
    - Invalid indicates that the page is not in the process' logical address space.

Shared pages (Slide 9.39) two implementations.

- **shared code**: one copy of read-only (reentrant) code shared among processes;
- **private code and data**: each process keeps a separate copy of the code and data.

Structure of Page Tables (Slide 9.41)

- **Hierarchical page table**: the page table may be too large, then we can split it into ***two-level page table*** (Slide 9.42) (a.k.a., forward-mapped page table); The original page number is divided into two parts: p_1 and p_2, which denotes the page number in the two levels of the page table. ***Three-level page table*** is also possible.
- **Hashed page tables** (Slide 9.47): common in address spaces > 32 bits; each element contains the virtual page number, the value of the mapped page frame, and a pointer to the next element.
- **Cluster page table** (Slide 9.47): variation for 64-bit addresses. Similar to hashed but each entry refers to several pages (such as 16) rather than 1; especially useful for sparse（稀疏的） address spaces.
- **Inverted page table** (Slide 9.48): use the physical page as indexes. One entry for each real page of memory.
    - Decrease memory needed to store each page table, but increase the searching time.
    - Use hash table to optimize the searching process!
    - Shared memory may need another mapping from a virtual address to the shared physical address.

Swapping (Slide 9.53): A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

- **Backing store**: fast disk large enough to accommodate copies 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; low-priority process is swapped out so higher-priority process can be loaded and executed.
- System maintain a ready queue of ready-to-run processes which have memory images on the disk (backing store).
- The swapped out process do not need to swap back into the same physical address most of the time (depend on address binding method).
- If the next processes to be put on CPU is not in memory, need to swap out a process a swap in target process, then the **context switch time** can be very high (because of accessing backing store, which is in the disk).
- ***Pending I/O***: can't swap as I/O would occur to wrong process.
  - *Solution*: Use ***double buffering***, first transfer I/O to kernel space, then to the destination (device / process).
- In modern operating system, we swap only when free memory is extremely low.

Two level pages: some computer such as Intel 32-bit architecture supports two page sizes of 4KB and 4MB.

Page address extension (PAE) (Slide 9.66) of Intel IA-32: allow 32-bit apps access to more than 4GB of memory space in all time; but in the certain time, it still can only access 4GB memory. Paging went to a 3-level scheme because the address is now 64-bit. If we do not want to change the page table size, then the length of page table can only be a half of the original length.

- 0~11: offset;
- 12~20: page table 1 (original: 12 ~ 21);
- 21~29: page table 2 (original 22 ~ 31).
- 30~31: page table 3 (page directory pointer table, CR3 register).

Chapter 10 Review

Virtual memory (Slide 10.7): separation of user logical memory from physical memory.

Virtual address space (Slide 10.8 ~ 10.10): the logical view of how process is stored in memory. Usually design logical address spaces for *stack to start at Max logical address and grow down* while *heap grows up*. Under the heap, we have data and code in order.

- Maximize address space use;
- Unused address space between the two is hole.

Demand paging: bring a page into memory only when it is needed.

- **Advantages**:
  - less I/O needed, no unnecessary I/O;
  - less memory needed;
  - faster response;
  - more users.
- Similar to paging system with swapping.
- **Lazy swapper**: never swaps a page into memory unless the page will be needed. Swapper that deals with pages is a pager.
  - With swapping, pager guesses which pages will be used before swapping out again; instead, pager brings in only those pages into memory;
  - If pages needed are already memory resident, no difference from non-demand-paging;

    - If page needed are not memory resident, then we need to detect and load the page into memory from storage;
        - without changing program behavior;
        - without programmers needing to change code;
- **Valid-invalid bit** (Slide 10.14): with each page table entry a valid-invalid bit is associated. (valid means in-memory, and invalid means not-in-memory).
    - If valid-invalid page table entry is i (invalid), then cause **page fault**.
    - **Handling page fault simple steps** (not consider that there is no free frames). A more completed one is in Slide 10.22 ~ 10.23.
        - If there is a reference to a page, first reference to that page will trap to OS and cause page fault;
        - OS looks at another table to decide: invalid reference, then abort; or just not in memory;
        - Find free frame;
        - Swap page into frame via scheduled disk operation;
        - Reset tables to indicate page now in memory, set valid-invalid bit to v (valid);
        - Restart the instruction that caused the page fault.
    - Hardware support: instruction restart, page table with valid-invalid bit, backing store.

Free-Frame list (Slide 10.21): a pool of free frames for satisfying such requests. OS typically allocate free frames using a technique known as ***zero-fill-on-demand***: the content of the frames zeroed-out before being allocated. When a system starts up, all available memory is placed on the free-frame list.

Performance of Demand paging (Slide 10.25)

$$EAT = (1-p)\cdot \text{memory access} + p \cdot (\text{page fault overhead} + \text{swap page in} + \text{swap page out})$$

Copy-on-Write (CoW) (Slide 10.27) allows both parent and child processes to initially share the same pages in memory. If either process modifies a shared page, only then the page is copied.

- More efficient process creation as only modified pages are copied.
- Free pages are allocated from a pool of zero-fill-on-demand pages.

Page replacement (Slide 10.31): prevent over-allocation of memory by modifying page fault service routine to include page replacement.

- Use modify (dirty) bit to reduce overhead of page transfers - only modified pages are written to disk.
- Page replacement completes separation between logical memory and physical memory; large virtual memory can be provided on a smaller physical memory!
- **Key**: select a victim frame.
- **Steps** (Slide 10.33):
    - Find the location of the desired page;
    - Find a free frame:
        - If there is a free frame, use it;
        - If there is no free frame, use a page replacement algorithm to select a victim frame. Write victim frame to disk if dirty.
    - Bring the desired page into the free frame; update the page and frames tables;
    - Continue the process by restarting the instruction that caused the trap.

    - **IMPORTANT:** potentially 2 page transfers (victim page write-back, and desired page write-in), so EAT increases.
- **Frame-allocation algorithms** (Slide 10.35) determines how many frames to give each process and which frame to replace;
- **Page-replacement algorithm** (Slide 10.35) wants lowest page-fault rate on both first access and re-access.
- **Reference string**: test string on different page replacement algorithms.
- **Algorithms**
    - **FIFO** (Slide 10.37) (queue algorithms).
    - **Belady's Anomaly**: adding more frames can cause more page faults.
    - **Optimal algorithm** (Slide 10.39) (stack algorithms).
    - **LRU algorithm** (Slide 10.40) (stack algorithms), add a counter for every page entry to implement.
    - **LRU approximation algorithm** (Slide 10.43): LRU needs special hardware and still slow.
        - **Reference bit**: with each page associate a bit, initially = 0. When the page is referenced, the bit set to 1; replace any reference bit = 0 (if one exists).
        - **Second-chance algorithm** Generally FIFO, plus hardware-provided reference bit. **Use clock replacement first**. If page to be replaced has reference bit = 0, then replace it; If the reference bit = 1, then set reference bit to 0 and leave the page in memory, replace the next page subject to the same rules.
        - **Enhanced second-chance algorithm** (Slide 10.45): improve the second-chance algorithm by using a reference bit and modify bit (if available) in concert. Take ordered pair (reference, modify) into consideration.
            - (0, 0): best page to replace;
            - (0, 1): not quite as good, must write out before replacement;
            - (1, 0) probably will be used again soon;
            - (1, 1) probably will be use again soon and need to write out before replacement.

          When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class. Might need to serach circular queue several times.
    - **Counting algorithms** (Slide 10.46): keep a counter of the number of references that have been made to each page, not common.
        - **Lease Frequently Used (LFU) Algorithm**: replaces pages with smallest count;
        - **Most Frequently Used (MFU) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
    - **Page-Buffering Algorithms** (Slide 10.47)
        - keep a pool of free frames. Read page into free frame and select victim to evict and add to free pool. When convenient, evict victim.
        - Possibly, keep list of modified pages. When backing store otherwise idle, write pages there and set to non-dirty.
        - Possibly, keep free frame contents intact and note what is in them.
            - If referenced again before re-used, no need to reload from disk, we can reload from trash!

        - Generally useful to reduce penalty if wrong victim frame selected.

Allocation of Frames (Slide 10.49) each process needs minimum number of frames, and maximum of course is total frames in the system.

- **Fixed Allocation** (Slide 10.50)
    - **Equal allocation**: equally give the processes the frames, keep some as free frame buffer pool.
    - **Proportional allocation**（按比例分配）: allocate according to the size of process.
- **Global replacement** (Slide 10.51): process selects a replacement frame from the set of all frames; one process can take a frame from another;
    - The process execution time can vary greatly;
    - Greater throughput so more common.
    - **Priority allocation** (Slide 10.73): get the victim frame from a process with lower priority number.
- **Local replacement** (Slide 10.52): each process selects from only its own set of allocated frames.
    - More consistent per-process performance;
    - Possibly underutilized memory.
- **Reclaiming Page** (Slide 10.52): a strategy to implement global page-replacement policy. All memory request are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin to selecting pages for replacement. ***Page replacement is triggered when the list falls below a certain threshold.*** This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.
- **No-Uniform-Memory-Access (NUMA)** (Slide 10.54) the speed of access to memory varies. Optimal performance comes from allocating memory close to the CPU on which the thread is scheduled.
- **Thrashing** (Slide 10.56): If a process does not have enough pages, the page-fault rate is very high, because it need page fault to get pages and replace existing frame, but quickly the replaced frame need to be replaced back. Leads to ***low CPU utilization***. Another definition: ***A process is busy swapping pages in and out***.
- **Demand paging & Thrashing** (Slide 10.58)
    - Why does demand paging work? Locality.
    - Why does thrashing occur? The summation of locality is larger than total memory size.
    - Limit effects by using local or priority page replacement.
- **Working-Set Model** (Δ) (Slide 10.60 ~ 10.64)

Allocating Kernel Memory (Slide 10.65): different from user memory, often allocated from a free-memory pool because some kernel memory needs to be contiguous, and kernel requests memory for structures of varying sizes.

- **Buddy System** (Slide 10.66) Allocates memory from a fixed-size segment consisting of physically-contiguous pages. Memory allocated using **power-of-2 allocator**.
    - Advantage: quickly coalesce（合并） unused chunks into larger chunk.
    - Disadvantage: fragmentation (especially internal).
- **Slab Allocator** (Slide 10.68)
    - Slab is one or more physically contiguous pages;
    - Cache consists of one or more slabs.
    - Single cache for each unique kernel data structures. Each cache filled with objects (instantiations of data structures). We try to put the object into the current slab, and if

    the current slab is full of used objects, then next object allocated from empty slab. If no empty slabs, new slab allocated.
  - Benefits include no fragmentation, fast memory request satisfaction.
  - **Simple Word Explanation**: combine several small data together and form a slab, then put the slab into the memory to reduce the memory waste (fragmentation).

Memory Compression (Slide 10.74): rather than paging out modified frames to swap space, we compress several frames into a single frame, enabling the system to reduce memory usage without resorting to swapping pages. (Compress the frames in the modified frame list, refer to Slide 10.47).

Pre-paging (Slide 10.76): to reduce the large number of page faults that occurs at process startup. Pre-page all or some of the pages a process will need, before they are referenced. But pre-paging loses may occur.

TLB Reach (Slide 10.78): the amount of memory accessible from the TLB. TLB reach = TLB size * page size. In order to increase TLB reach, we can ***increase page size*** or provide multiple page sizes.