Chapter 6

- **Race condition:** multiple processes access and manipulate the same data concurrently, and the result is dependent on the particular execution order
- **Critical section:** a segment of code, in which a process can access resources that are shared with other processes
- **Three requirement:** M.E; Progress (deadlock/livelock; entering condition); Bounded waiting (starvation)
- **Kernel expose to race condition:** example pid() both processes trying to access the next_available_pid variable
- **non-preemptive kernels:** free from race conditions, easier to design (don't need to design mechanism for context-switch)
- **Preemptive:** more responsive, and enables interactivity, however, may have data inconsistency issues
- **Memory barrier:** same idea as midterm short answer (visibility)
- **Mutex:** to ensure one process can go into the C.S.
- **Semaphore:** can be counting, can be mutex (int value = 1)
    - With busy waiting
    - Without busy waiting
- **Monitor:** monitor is an abstract **data type**, with a set of defined operations that guarantees **M.E** within the monitor
    - Implementation of monitor using semaphore*
    - Signaling mechanisms
        - Signal and continue (logical condition may not hold after the signalling process finishes execution)
        - Signal and wait (with interruption)
        - Signal and quit (compromise)

- **Peterson's:** 2 variables (turn and flag[i])
    - Flag is the boolean, turn is the integer
    - Busy waiting condition for process i: flag[j] == true && turn == j

```
// Peterson's solution with two variables;

// Peterson's solution
flag[i] = true;
turn = j;
while(flag[j] && turn = j);
// CS
flag[i] = false;
// RS
```

```
// Peterson's solution w/ flag

flag[i] = true;
while (flag[j]);
//CS
flag[i] = false;
//RS
```

```
// Peterson's Solution w/ int

int turn = 0;
//For process i:
while (true) {
  while (turn == j); //busy wait, entry section
  //critical section
  turn = j; // exit section
  //remainder section
}
```

- **Limitations for peterson's solution:**
    - During compiling time, the order could not be guaranteed (modern computer architecture)
    - Only works with two processes

—---

Chapter 5 CPU scheduling

- CPU scheduling decisions may take place when a process
1. **Switches from running to waiting**
2. switches from running to ready state (during an interrupt) (maybe)
3. from waiting to ready (completion of an I/O) (maybe)
4. **terminations**
5. from new to ready

In a non-preemptive environment, only 1,4 can happen

And during those two processes: a new process MUST be selected for scheduling

Scheduling Criteria: cpu utilization, throughput, turnaround time, waiting time (time spent in ready queue), response time

Reason why we use waiting time: independent of time spent in the blocking states

—-

Chapter 8 Deadlocks

- What is Deadlock: a **waiting** process cannot **change** its current state because one or more resources that it has requested are currently held by other processes, who are in turn waiting for a resource being held by the first process
- DDL characterizations
    - M.E
    - Hold and Wait
    - Non-preemption
    - Circular wait
    - The above four conditions have to all present for a DDL to potentially happen

    - Necessary but not sufficient
- Resource allocation graph
    - 3 edges (request, assignment, claim)
    - 2 vertices (resource, threads/processes)
    - **If each resource only has 1 instance**, cycle(s) in the graph indicates deadlock (100%)
- Four different ways to deal with DDL
    - Ignore
    - DDL prevention: break one of the above conditions
    - DDL avoidance: using additional information from the system/process to make decisions regarding resource allocation
    - DDL recovery: detection algorithm + recovery
- Prevention:
    - M.E. there is nothing too much we can do
    - Hold and wait
        - A process can run only if all resources are allocated
            - Low resource utilization
        - Release all the resources it currently holds when it makes a request
            - Starvation
    - No-preemption:
        - If a process is trying to get a resource that is not available, so it preempts all resources it currently holds
            - Possible starvation
    - Circular wait:
        - We need to come up with a certain order, so the ordering can only go up
        - If the resource the process is requesting has a lower order, the process has to start over
            - Possible starvation
            - Very hard to implement, so not practical
    - Everything here is not practical, that’s why we don't do preventions
- DDL avoidance
    - safe/unsafe state/deadlock state
        - OS will be able to control the transition from safe to unsafe
        - There is no control from unsafe to deadlock state, DDL state is a sub-state of unsafe state
    - Safe sequence: OS can allocate resources to different processes in a particular order to avoid entering unsafe state, this order is called safe sequence
        - If there is no safe sequence, there is no safe state
    - OS can control if it is entering the unsafe by controlling the resource allocation
- Two schemes for DDL avoidance:
    - Resource allocation graph (not generalized solution, we assume one instance for each resource)
        - We will just make sure granting a request will not lead to cycle in the graph

        - Claim edge: the request is pending, and might be fulfilled in the future, and we also take claim edge into the consideration for allocating resources
        - Transitions (for claim edge):
            - When a request is made, claim edge -> request edge
            - When a request is granted, request edge -> assignment edge
            - When the resource is released, assignment edge -> claim edge
        - Complexity: O(n^2) (n is the number of threads)
        - The cycle with claim edge: may or may not be a deadlock
    - Bankers:
        - Generalized
        - 4 different data structures m resources, n processes
            - MAX (n*m matrix)
            - Need (n*m)
            - Allocation (n*m)
            - Available (m vector)
        - Safety algorithm
            - Finished(n), boolean, indicates the status of each process
            - Work = available
            - Find a process where finish[i] = false, and need[i] < Available
                - Work = available + allocation[i]
                - Finish = true
            - If all finish = true, there is a safe sequence
            - Complexity: O (m* n^2)
            - step1 complexity O(m), step4: O(n)
        - Resource allocation (request)
            - Request[i] < Need
            - Request[i] < Available
            - Make change to corresponding data structures (pretend granting the request) (3 data structures updated)
                - Available = Available - Request
                - Allocation = Allocation + Request
                - Need = Need - Request
            - Call the safety algorithm
            - O(m*n^2)
        - Not practical: calculation is expensive, and each process has to pre-determine the maximum it needs during execution
- Detection
    - Wait-for graph, if ti $\rightarrow$ tj, means ti is waiting for a resource that is currently held by tj
        - One instance per resource, so cycle = DDL
        - O(n^2)
        - Invoked once in a while (with different thresholds)

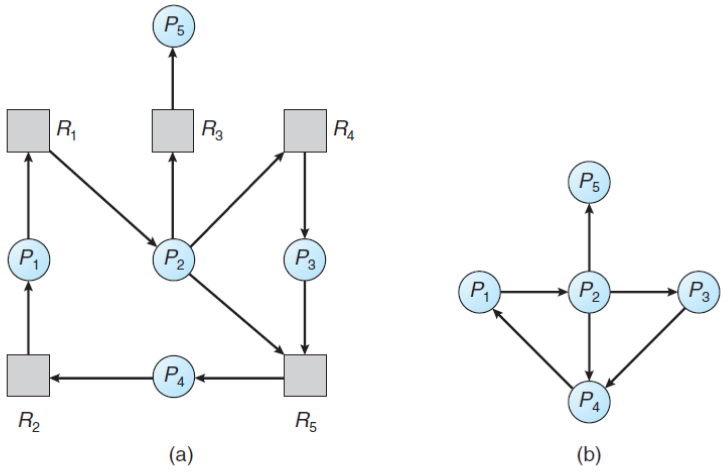

Figure 7.9 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

- Similar to banker's algorithm, but with different data structures
    - For determining finish status: 0 allocation = irrelevant to DDL
    - Available, Allocation, Request
    - Only comparing request and work (allocation + available)
    - O(m*n^2)
- Recovery:
    - Abort all processes that are deadlocked
        - Computation power waste
    - Abort one process at a time until there's no DDL
        - Run detection each time (overhead)
        - Moving data between register and memory frequently causes a huge overhead
    - To minimize cost
        - Priority
        - Elapsed time
        - Amt of resources (used and need)
    - Issues to be addressed
        - Select a victim
        - Rollback: roll back to safe state
        - Starvation: how to not keep killing the same victim (nb of rollbacks)

Chapter 9 Memory Management

- Memory: consists of a huge array of bytes
- Two things we care about: speed + protection
    - Speed can be achieved by cache
    - protection there are different ways, we will explore them together!
- 1st protection mechanism (base + limit)
    - Do you remember the graph?

- Address binding:
    - What is address binding: it is a translation: from logical to physical
- Different stages that address binding can happen
    - Loading and runtime are relocatable, runtime binding allows the process physical memory address to change dynamically
- Relocatable binding: MMU generates relocatable address (relocatable register)
- DLL
    - One copy in memory, multiple processes can have access to it
    - Saves space
    - Update itself, easier version control
- Contiguous memory allocation:
    - Protection is achieved with a relocation register (remember another graph right?)
    - How to track memory usage (of course by the OS)
    - 3 different approaches for dynamic allocation
        - Which the most simple one: first, and the best time-complexity
        - Which the worst one: worst fit
    - Issue: external fragmentation 50% rule, clusters of blocks
        - Compaction (time consuming, and not all processes can be moved)
        - Paging (allows non-contiguous allocation)
        - Page- frames (1,1)
- Paging:
    - Paging provides a separation between physical address and logical address
    - Logical address space, page entry, page size, offset, a bunch of calculations (from assignment 3 Q6)
    - Translation from MMU (page number, frame number, offset)
    - Page table builds the connection between frame number and page number
    - Issue with paging: internal fragmentation, slow memory access
        - Solve: smaller page size (more entries, large page table)
        - TLB
    - When a process comes into the system, size will be checked, loaded in, build a page table for this process, everything is pretty much done by the OS
    - Programmer view: contiguous, physically: scatter AF
    - Free frame list: data structure maintained by the kernel, to keep track of free space
- Page table (pointer in PCB: PTBR)
    - When process resumes execution, it must reload user register + contents according to page table
    - Can be implemented as registers
    - Can reside in memory
        - Slooooowwwwww
    - PTBR changes whenever there is a context switch
    - TLB: fast memory, small size
        - Issue with TLB flash everytime, to solve: ASIDs
    - EAT calculation with TLB

    - Memory protection with paging:
        - Protection bits
        - PTLR
- Paging for sharing: shared pages for reentrant code, different logical addresses map to the same physical address
- Size issue with page table
    - Hierarchical
        - Issue: + 100% memory access time with each hierarchical structure
    - Hashed page table
        - With hash function applied to page number
        - Entry can be chained
        - Issue: fixed hash table size
            - Solve: clustered page tables
    - Inverted page table
        - 1 table in the OS
        - Less memory needed, longer search time
        - Sharing is gonna be challenging

Chapter 10 Virtual Memory

OBJ : allow the execution of a process not completely in memory

+ : programs can be greater in size than physical memory, processes can share files and it is an efficient mechanism for process creation
- : not easy to implement and also decrease general efficiency (page fault rate)

Pure demand paging

- Load pages when they are needed (like swapping) (and only when they are required)
  → we need to know what pages are in memory and on disk → use valid (legal + memory) or invalid (not valid or valid + disk) bit scheme in page table
  - If a process tries to access an invalid page → **page fault** (interrupt and trap to the OS) → we need to save the current states and page in
  - EAT = (1-p) * ma + p * t_pf (ma = memory access time ~200ns and t_pf ~8ms and p is the probability of page fault)

THERE ARE TECHNIQUES FOR PAGE REPLACEMENT

However, we have to keep in mind that filling a great number of frames will lead to over allocation

Use modify=dirty bit to see if the page has been modified or not

If we need to page in, and there are no free frames, then we need to find a victim

Frame replacement algorithms

FIFO		Important : in this scheme, having 4 frames is worst than having 3 → **Belady's anomaly**
Optimal Page replacement		Replace the page that **will** not be used for the longest period of time BUT we cannot know the future
Last recently used (LRU) Counter -> overflow the clock Stack and LRU at the bottom	LRU approximation	Use a **reference bit** to know if a page has been used and refresh it at regular intervals
	2nd chance	FIFO then check reference bit : 0-> replace and 1 -> spare and set to 0 (circular queue)
	Enhanced second-chance	Use **tuple (reference bit, modify bit)**
	Counting-Based page replacement	Least Frequently Used (LFU) : shift the count sometimes Most Frequently Used (MFU) : based on the hypothesis that a page with the smallest count could also be new in memory (just been brought)
	Page Buffering	Use a **pool of free frames** so a new page is read in FF while victim is written out and its frame is declared a FF

Frame allocation algorithms

Pb : how to allocate the fixed amount of free memory ? → split m frames among n processes
The min number of frames is defined by the computer architecture
How to split m frames among n processes ?

Equal allocation	Each of the processes has **m/n** frames
Proportionnal allocation	If p_i has a virtual size of s_i and S = sum(s_i) then p_i has **s_i/S** frames

HOWEVER : high and low priorities are treated the same so the better would be to use a combination of size and priority

Global allocation More commonly used	Process selects replacement frame from the set of all frames
Local allocation	Each process selects from its own set of allocated frames

Non-uniform memory access (NUMA) : multi CPU → a CPU can access some sections of main memory faster than it can access other (CPU and boards interconnected)
BUT slower than if same board

Thrashing : high paging activity

Number of frames allocated to a low priority priority < threshold (computer architecture) → suspend process OTHERWISE a lot of page faults

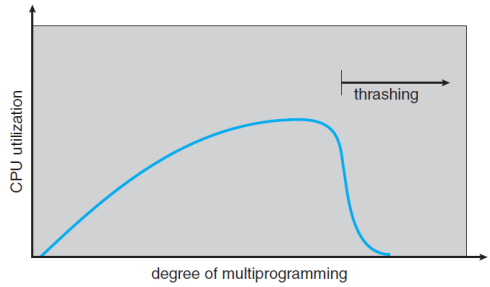

Figure 9.18 Thrashing.

Limit the effect :
Use local/priority replacement so one thrashing process will not cause other processes to thrash BUT thrashing processes will be in the waiting queue for paging device

WORKING-SET MODEL :
Use a working-set window to examine the + recent page references

PAGE FAULT FREQUENCY

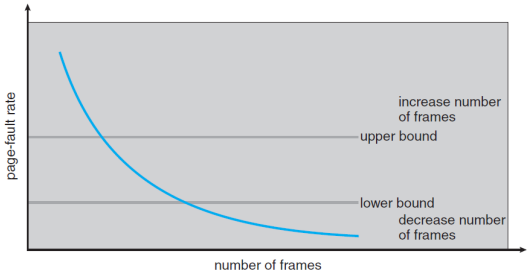

Figure 9.21 Page-fault frequency.

—--
Chapter 14: File System Implementation

File stored on secondary storage ⇒ Holds large amount of data
Use of disks → Can access all the information stored on it
I/O transfers between **memory** and **disk** is done in blocks

Structures:
Boot control block: Infos to boot system
Volume control block: Details about nb of blocks, size of blocks, free-blocks count,...
Directory structure: To organize files
Per-file FCB (File Control Block): Details about the file
Mount table: Infos about each mounted volume
Directory-structure cache: directory infos of recently accessed directories
System-wide open-file table: Contains FCB of each open file
Per-process open-file table: Pointers to appropriate FCB in system-wide Open-FIle table + other info
Buffer: Hold blocks while in use

Creating new file:
- Call logical file system
    - Allocates new FCB
    - Reads appropriate directory into memory
        - Updates directory with new filename & FCB
        - Writes back to file system

Open() and Close() system calls:
- Open(): Search **system-wide OFT** to see if file is open
    - If not, put FCB into the table
    - Then put entry in **per-process OFT**
- Close(): Remove **per-process OFT** entry
    - Decrement **system-wide OFT** “open count”
        - If count = 0, remove **system-wide OFT** entry

Directory implementation:
1. Linear List:
   Linear list of file names with pointers to data blocks
   Pb: Search is linear (have to read through the list)
2. Hash table:
   No linear search, but table and hash function have fixed size. Need to update every entry to change size
   But : Can use linked-list as entries to circumvent size issue

Allocation Methods:
- Contiguous:
    - Each file occupies set of contiguous blocks

    - Disk: Minimal number of “disk seek” to access contiguous files
    - Is defined by: The Address of the 1st block and the length of the file (similar to base+limit from paging)
    - **Advantages:** Easy to access the next block or any block
    - **But:** Difficult to find space for a new file + fragmentation + Problem of choosing the right size for a new file
- Modified Contiguous:
    - Add another chunk of contiguous space (called an **extent**)
- Linked allocation:
    - Is a linked list
    - Directory contains a pointer to the first and last blocks
    - **Problem:** Only allows sequential search + **need space** for pointers
        - **Solution:** Use clusters of blocks (like pages) ⇒ Allocate clusters rather than blocks. Can cause internal fragmentation if cluster is not fully used
    - **Big issue:** If a pointer is lost (cuts the whole list)
        - Solved with a **file-allocation table**:
            - One entry per block
            - Indexed by the block number
            - The **directory entry** has the number of the first block
            - A **block entry** has the number of the next block
- Indexed Allocation:
    - One index block per file ⇒ it contains an array of all the other **storage-blocks** addresses
    - Similar to a page table
    - **Problem:** Pointer overhead is greater than for Linked Allocation
    - Linked Scheme: Index block has the address of storage blocks and of the next index-block
    - Multi-level-index = Multi-level paging. One index block that contains pointers to other index blocks (who contain pointers to storage blocks, or other index for a third level)
- Combined Scheme:
    - Contiguous for small file size then switch to linked or indexed when size grows
- Performance:
    - Need to determine how systems will be used to choose the allocation method
    - Linked allocation does not allow direct access