# CMSC412: Operating Systems

Michael Li

## 1 Deadlocks

### 1.1 Overview

Threads utilize resources in the following sequence:

- **Request**: if request cannot be granted immediately (e.g. mutex lock is held by another thread), then the requesting thread must wait until it can acquire the resource
- **Use**: thread operates on the resource (e.g. thread is allowed to access critical section)
- **Release** thread releases the resource

Requests and releases can be done as system calls (e.g. `open()` and `close()` a file, or `allocate()` and `free()` memory)

System table records whenever each resource is free or allocated, and the thread to which it was allocated to.

- If a thread requests a resource allocated to another thread, it will be added to a wait queue for that resource

Set of threads is **deadlocked** when every thread in it is waiting for an event that can only be caused by another thread in the set

- Usually resource acquisition and release

### 1.2 Deadlock in Multithreaded Applications

Let `pthread_mutex_int()` create an unlocked mutex, and `pthread_mutex_lock()` and `pthread_mutex_unlock()` acquire and release the mutex. Then

```
/* thread one runs in this function */
void *do work one(void *param) {
  pthread mutex lock(&first mutex); pthread mutex lock(&second mutex);
  /**
   * Do some work
   */
  pthread mutex unlock(&second mutex);
  pthread mutex unlock(&first mutex);

  pthread exit(0);
}

/* thread two runs in this function */
void *do work two(void *param) {
  pthread mutex lock(&second mutex);
  pthread mutex lock(&first mutex);
  /**
   * Do some work
   */
  pthread mutex unlock(&first mutex);
  pthread mutex unlock(&second mutex);
```

```
  pthread exit(0);
}
```

can cause a deadlock if `thread_one` acquires `first_mutex` while `thread_two` acquires `second_mutex`. However, this is not guaranteed to happen each run

**Livelock** occurs when threads are unable to proceed for different reasons

- Deadlock occurs when every thread is waiting for an event that can only be caused by another thread
- Livelock can occur when a thread continuously attempts an action that fails

## 1.3 Deadlock Characterization

Deadlock can arise if all 4 conditions hold simultaneously

1. **Mutual Exclusion**: at least one resource must be held in nonsharable mode (if another thread requests it, the requesting thread must be delayed until the resource has been released)
2. **Hold and wait**: thread must hold at least 1 resource and be waiting to acquire resources held by other threads
3. **No preemption**: resources can't be preempted
4. **Circular wait**: $T_0$ is waiting for resource held by $T_1, \dots T_n$ waitinf ro resource held by $_0$

Information is maintained in a **resource-allocation graph** where $V$ is the set of active threads and resource types

- $T_i \to R_j$ signifies that $T_i$ has requested an instance of $R_j$
- $R_j \to T_i$ signifies that an instance of $R_j$ has been assigned to $T_i$
- If no cycle exists, then thread CANNOT have a deadlock
- If cyclic exists, then thread MAY OR MAY NOT have a deadlock (each resource type can have several instances)

Deadlocks can be dealt by:

- Ignoring it (low risk of occurrence of deadlock isn't worth cost of constantly checking)
- **Prevention**: ensures that at least 1 of the necessary conditions canot hold
- **Avoidance**: operating system is given additional information in advance about what resources a thread will request and decides for each request if the thread should wait
- **Recovery**: allow deadlock to occur and then use an algorithm to recover from deadlock

## 1.4 Deadlock Prevention

Idea is to ensure that at least 1 of the deadlock conditions cannot hold

- **Mutual Exclusion**: not all resources need to be nonsharable (e.g. read-only files)
- **Hold and Wait**: enforce that when a thread requests a resource, it cannot hold any other resources.
  - Require threads to request and be allocated all of its resources before beginning execution
  - Only allow a thread to request resources when it has none. A requesting thread can release all of current resources.

    Both of these can suffer from low resource utilization (resource allocated but unused for a long time) and **starvation** (thread that needs popular resources may have to wait indefinitely)
- **No Preemption**:
  - If a thread is holding resources that another thread wants, then all resources in the current thread is preempted (released). The current thread will only be restarted once it can regain its old resources
  - If a thread requests resources, check if they are available. If so, we allocate them. Otherwise, check if they are allocated to some other thread that is waiting for more resources. If so, we preempt the desired resources from the waiting thread. If the resources are not available and not held by a waiting thread, the requesting thread waits. It can only be restarted once it is allocated the desired resources, and any resources that were preempted while waiting.

  Usually can't be applied to locks and semaphores, so not too useful for deadlocks

- **Circular Wait**: impose a total ordering on all resources types. Threads can only request resources in an increasing order of enumeration
  - After a thread has requested for resource $R_i$, then it can only request for $R_j$ if and only if $F(R_j) > F(R_i)$
  - If several instances of the same resource type are needed, then a **single** request for all of them is issued

## 1.5 Deadlock Avoidance

Deadlock prevention can lead to low device utilization and reduced system throughput. Deadlock avoidance examines resource-allocation state to ensure that a circular-wait can never exist, although this can result in lower resource utilization

**Safe State**: system can allocate resources to each thread in some order and still avoid deadlock (can only occur if a **safe sequence** exists)

- resource requests from $T_i$ can be satisfied by the current available resources and any resources held by $T_j$ with $j < i$

### 1.5.1 Resource Allocation Graph Algorithm

For systems where we only have one instance of each resource type, we can add **claim edge** $T_i \rightarrow R_j$ to indicate that $T_i$ may request $R_j$ in the future

- When the request is initiated, the claim edge is converted to a request edge
- When $R_j$ has been released by $T_i$, the assignment edge is converted to a claim edge
- Request is granted only if the resulting assignment edge DOES NOT result in a cycle. If so, $T_i$ will have to wait

### 1.5.2 Banker's Algorithm

When a thread enters the system, it declares the maximum number of instances of each resource type it needs

- When a user requests a set of resources, the system checks if it will result in a safe state. If so, the resources are allocated. Otherwise, the thread waits until other threads release enough resources

Makes use of several data structures for $n$ threads and $m$ resources types

- **Available**: vector of length $m$ indicating number of available resources types
- **Max**: $n \times m$ matrix defining maximum demand of each thread
- **Allocation**: $n \times m$ matrix defining number of resources of each type currently allocated to each thread
- **Need**: $n \times m$ matrix indicating remaining resource needs for each thread
- `Need[i][j] = Max[i][j] - Allocation[i][j]`

**Safety Algorithm**: check if system is in a safe state $O(m \times n^2)$

1. Let `Work = Available` and `Finish = false` be vectors of length $m$ and $n$
2. Find an index $i$ such that `Finish[i] == false` AND `Need_i <= Work`. If no $i$, skip to step 4
3. `Work = Work + Allocation_i`

   `Finish[i] = true`

   Goto step 2
4. If `Finish[i] == true` for all $i$, then the system is in a safe state

**Resource-Request Algorithm**: determine whether requests can be safely granted

1. Let `Request_i` be the request vector for $T_i$.
2. If `Request_i <= Need_i` go to step 3. Otherwise raise an error
3. If `Request_i <= Available`, go to step 4. Otherwise, $T_i$ must wait

4. Have the system pretend to have allocated request resources to $T_i$ by doing

   `Available = Available - Request_i`

   `Allocation_i = Allocation_i + Request_i`

   `Need_i = Need_i - Request_i`

   If this results in a safe state, the transaction is completed and $T_i$ is allocated. Otherwise $T_i$ must wait for `Request_i`

## 1.6 Deadlock Detection

For single instance resource types, can use a **wait-for** graph, identical to the resource-allocation graph except only without resource nodes

- $T_i \to T_j$ implies that $T_i$ is waiting for $T_j$ to release a resource that $T_i$ needs
- Deadlock exists if and only if the wait-for graph has a cycle

For several instance resource types, a variation of Banker's Algorithm

- **Available**: vector of length $m$ indicating number of available resources types
- **Allocation**: $n \times m$ matrix defining number of resources of each type currently allocated to each thread
- **Request**: $n \times m$ matrix indicating the current request of each thread

**Detection Algorithm**: $O(m \times n^2)$

1. Let `Work = Available` and `Finish = false if Allocation_i != 0 else true` be vectors of length $m$ and $n$
2. Find an index $i$ such that `Finish[i] == false` AND `Request_i <= Work`. If no $i$, skip to step 4
3. `Work = Work + Allocation_i`

   `Finish[i] = true`

   Goto step 2
4. If `Finish[i] == false` for some $i$, then the system is deadlocked

## 1.7 Recovery from Deadlock

### 1.7.1 Process/Thread Termination

- Can abort all deadlocked processes, although this is expensive
- Abort 1 process at a time until the deadlock cycle is eliminated (overhead of checking for deadlock)

Need to ensure that terminating the process doesn't leave data in an incomplete state. Also need to choose the **minimum costing** process to terminate

### 1.7.2 Resource Preemption

Preempt some resources from processes and give them to other processes until the deadlock cycle is broken. Need to overcome 3 issues:

1. **Selecting a victim**: need to determine of preemption to minimize cost
2. **Rollback**: if we preempt a resource from a process, we need to roll back the process to some safe state and restart it from that state
3. **Starvation**: ensure that starvation doesn't occur (resource gets preempted from the same process)

# 2 CPU Scheduling

## 2.1 Overview

Goal of **multiprogramming** is to load multiple programs into memory and have some process running at all time

- When CPU becomes idle, the **CPU scheduler** selects the next process in memory to execute and allocates the CPU to that process

Process execution consists of **cycles** of CPU execution and I/O wait (alternating between **CPU burst** and **I/O burst**)

CPU-Scheduling decisions may take place during the following four circumstance:

- When a process switches from running to waiting (e.g. I/O request)
- When a proces switches from running to ready (e.g. interrupt occurs)
- When a process witches from waiting to ready (e.g. I/O completed)
- When a process terminates

Scheduling can either be **nonpreemptive** or **preemptive**

- Preemptive system can lead to race conditions and may leave important data in an incomplete state
- For interrupts, the system will need to disable and reenable interrupts during the critical sections to ensure that concurrent access doesn't cause issues

**Dispatcher** gives control of the CPU's core to the process selected by the CPU scheduler. This involves

- Switching context of processes (creates **dispatch latency**)
- Switching to user mode
- Jumping to the proper location in user program to resume execution

## 2.2 Scheduling Criteria

- **CPU utilization**: keep CPU as buys as possible
- **Throughput**: number of processes completed per time unit
- **Turnaround time**: how long it takes to execute a process (time in waiting queue, execution, and doing I/O)
- **Waiting time**: how long process spends waiting in the ready queue
- **Response time**: time from request submission until the first response is produced

## 2.3 Scheduling Algorithms

### 2.3.1 First-come, First-served (FCFS)

Managed using a FIFO queue. When the CPU is free, it is allocated the process at the head of the queue

Average waiting time can be long if the first process has a long CPU burst time. All other processes may have short CPU burst times but have to wait for a long time

- **Convey effect**: all other processes wait for one big process to get off the CPU, resulting in lower CPU and device utilization. Can be resolved by allowing the shorter processes run first

FCFS scheduling is **nonpreemptive** since process keeps CPU until it releases the CPU (termination of I/O request). Thus FCFS can be troublesome for interactive system where each process needs to share the CPU at regular intervals

### 2.3.2 Shortest-Job-First (SJF)

CPU is assigned to the process with the smallest next CPU burst

Provably optimal but impossible to implement since we don't know the length of the next CPU (have to predict length)

Next CPU burst is predicted as an **exponential average** of the previous CPU bursts $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

- $0 \leq \alpha \leq 1$
- $t_n$ is the length of the $n$th CPU burst
- $\tau_{n+1}$ is the predicted value of next CPU burst

SJF can either be **preemptive** or **nonpreemptive**

- For preemptive, if the next CPU burst is shorter than what is left of the currently executing process, the current process is preempted

### 2.3.3 Round-Robin

Similar to FCFS but preemption is enabled and each process is given a **time quantum** before it is preempted by the next process in the ready queue

Often results in long average waiting time

Turnaround time also depends on size of time quantum (want to have next CPU burst finish in a single time quantum)

### 2.3.4 Priority Scheduling

Each process is associated with a **priority** and CPU is allocated to the next process with the highest priority

Priority Scheduling can be **preemptive** or **nonpreemptive**

- For preemptive, if the next process's priority is higher than the priority of the currently running process, the current process is preempted
- For nonpreemptive, the new process is placed in the ready queue, sorted by priority number

Priority Scheduling can lead to **starvation** where a process is ready to run but can't be allocated the CPU due to a low priority number

- One solution is to use **aging** where the priority of a process that is waiting gradually increases
- Another solution is to combine round-robin and priority scheduling where the system executes the highest-highest-priority process and runs processes with the same priority using round-robin

### 2.3.5 Multilevel Queue

Have a separate queue for each distinct priority and priority scheduling simply schedules a process in the highest-priority queue

- Can be combined with round robin to have multiple highest-priority process run
- Also could give each queue a certain portion of CPU time to prevent starvation

### 2.3.6 Multilevel Feedback Queue

Under multilevel queue, processes are permanently assigned to a queue. In contrast, multilevel feedback queue allows processes to move between queues

- processes that use too much CPU time are moved to lower priority queue
- process that waits too long in a lower-priority queue can be moved up to a higher-priority queue (aging to prevent starvation)

## 2.4 Thread Scheduling

Typically only kernel-level threads are scheduled by operating system. User-level threads are usually managed by a thread library and the kernel is unaware of them. To run, user-level threads must be mapped to an associated kernel-level thread (usually through LWP)

### 2.4.1 Contention Scope

For many-to-one and many-to-many thread models, the thread library schedules user-level threads to run on an available LWP through **process contention scope (PCS)**. To decide which kernel-level thread to schedule onto CPU, the kernel uses **system-contention scope (SCS)**

- Note that PCS will typically preempt a thread currentlyr unning in favor of a higher-priority thread

## 2.5 Multi-Processor Scheduling

If multiple CPUs are available, **load sharing**, where multiple threads can run in parallel, is available

- Makes scheduling more complex

### 2.5.1 Approaches to Multi-Processor Scheduling

- **Asymmetric multiprocessing**: only one processor handles scheduling decisions, which reduces the need for data sharing but can easily become a bottleneck
- **Symmetric Multiprocessing (SMP)**: each processor is self-scheduling where each threads selects a thread to run, improving performance but workloads might be of variable sizes

### 2.5.2 Mutlicore Processors

A single processor might have multiple cores. This results in faster operations but may complicate scheduling

**Memory stall**: processor spends a significant amount of time waiting for data to become available. Occurs because processor operates faster than memory does or because of a cache miss. Various solutions exist

- **hardware threads**: if one hardware thread stalls, then the core can switch to another thread 0 **chip multithreading (CMT)**: each hardware thread in a core is treated logically as its on CPU

Two ways to multithread a processing core

- **coarse-grained**: a thread executes until a long-latency event (e.g. memory stall) occurs. The core switches to another thread but cost of switching betweeng threads is high
- **fine-grained**: switches between threads at a finer level of granularity (e.g. instruction cycles). Cost of switching threads is small because of logic architecture

  Important to note that resources of core (e.g. caches and pipelines) must be shared between hardware threads

### 2.5.3 Load Balancing

Important to balance workload on SMP systems between processors so that one processor doesn't sit idle

- **push migration**: specific task taht periodically checks the load of each processor. It will evenly distribute the load by moving threads from overloaded to less-busy processors.
- **pull migration**: idle processor pulls a waiting task from a busy processor

### 2.5.4 Processor Affinity

For a thread running on a specific processor, it will populate the cache of the processor and successive memory access by the thread are often satisfied in cache memory (**warm cache**). However, if the process is migrated to another processor, the cache of the first processor must be invalidated and the cache of second processor must be repopulated

- This operation is costly

**Processor affinity**: process has an affinity for the processor it is currently running

- For a common ready queue, a thread may be selected for execution by any processor. Thus we may need to repopulate processor cache
- For per-processor queue: a thread can only be scheduled by the same processor and thus results in warm cache and processor affinity

Forms of procesosr affinity

- **soft affinity**: system attempts to keep process running on safe processor but doesn't guarantee
- **hard affinity**: process specifies a subset of processors for which it can run on

### 2.5.5 Heterogenous Multiprocessing (HMP)

Systems designed using cores that can run the same instructions but vary in terms of power management. Goal is to manage power consumption by assigning tasks to certain cores based on task demands

## 2.6 Real-Time CPU Scheduling

**Soft real-time systems** provide no guarantee to when a critical real-time process will be scheduled

- Only guarantees that real-time process will be given preference over noncritical processes

**Hard real-time systems** enforce that a task must be serviced by a deadline

### 2.6.1 Minimizing Latency

Want to minimize **event latency**: time from when the event occurs to when it is serviced

**Interrupt latency**: time from arrival of an interrupt to the start of the service routine (system finishes its current instruction and determine the type of interrupt)

**Dispatch latency**: time required for scheduling dispatcher to stop a process and start another process

### 2.6.2 Priority-Based Scheduling

We want to respond immediately to real-time processes. Thus a priority-based algorithm with preemption must be used

- Doesn't guarantee hard real-time task functionality since we also need to meet a deadline requirement

Hard real-time can be guaranteed using **admission-control** where the scheduler admits process with tasks it can finish on time and rejects processes with tasks it cannot finish on time

### 2.6.3 Rate-Monotonic Scheduling

Schedules periodic tasks using a static priority policy with preemption. Process's with lower **period** have higher priority

- **Period**: process requires CPU at fixed periods/intervals

### 2.6.4 Earliest-Deadline-First Scheduling (EDF)

Priorities are dynamically assigned according to deadliness. Each process must deadline requirements to the system

### 2.6.5 Proportional Share Scheduling

Allocates $T$ shares among all applications. Each application can receive $N$ shares of time, so each application will have $N/T$ of total processor time

# 3 Main Memory

## 3.1 Overview

CPU can only directly access registers or main memory

- Register content can be accessed in 1 CPU cycle
- Main memory usually takes many cycles (CPU **stalls** until data is acquired).
- CPU uses **cache** for fast access to memory

Each process should have a separate memory space to protect processes from each other

- Address are determined by **base register** and **limit register**, which can only be loaded by operating system using a privileged instruction

**Address binding**, converting logical addresses to physical addresses, can be done during different times:

- **Compile Time**: if you know where the process will reside in memory at compile time, **absolute code** can be generated. Requires recompilation if starting location changes
- **Load Time**: compiler can generate **relocatable code** where binding is delayed until load time
- **Execution Time**: process can be moved during its execution to a different memory segment, so binding must be delayed until run time

Compile or load time binding results in identical logical and physical addresses. Execution time binding results in differing logical and physical addresses (handled by **memory-management unit (MMU)**)

- **Logical address space**: set of all logical address generated by a program

- **Physical address space**: set of all physical addresses corresponding to logical addresses

**Dynamic Loading**: routine is not loaded until it is called and removed when it is done, ultimately minimizing the amount of in memory use

## 3.2 Contiguous Memory Allocation

Memory is divided into 2 partitions: one for the operating system (usually high memory) and one for user processes

- **Contiguous memory allocation**: each process is contained in a single contiguous section of memory

For a **variable-partition scheme**, the operating system keeps a table indicating which parts of memory are available. If a process request comes but we don't have enough memory, we can

- ignore the request
- add the process to the wait queue

Different models exist for how to allocate chunks of memory:

- **First-fit**: find the first hole that is big enough to allocate
- **Best-fit**: allocate the smallest hole that is big enough
- **Worth-fit**: allocate the largest hole

Dynamic allocation can result in **fragmentation**

- **External fragmentation**: free memory is broken into little pieces but can't be combined to service the next request
  - One solution is to use **compaction** where we shuffle memory content so all the free memory form one large block. Cannot be done if relocation is static or done at load time
  - Another solution is to permit logical address space to be noncontiguous (**paging**)
- **Internal fragmentation**: we allocate extra memory for a request to minimize number of holes and thus have unused memory in that partition

## 3.3 Paging

Allows process's physical address space to not be contiguous, avoiding external fragmentation but can still result in internal fragmentation

Physical memory is broken into fixed-sized blocks calledd **frames** and logical memory is broken into same size blocks called **pages**

- A process's page is loaded into memory frames when it is executing
- CPU address is divided into **page number** and **page offset**
- Page number indexes per-process **page table** that has the base address of each frame

For paging, there is a clear separation between programmer's view of memory and the actual physical memory

- The programmer sees memory as one single contiguous space with only one program
- User program is actually scattered throughout physical memory that also holds other programs

Operating system also manages a **frame table** that stores information about which frames are allocated and which are available

For machines with large page tables, page tables are kept in main memory and a **page-table base register (PTBR)** points to the page table and can change page tables accordingly

Processes rarely use all of its address ranges, so systems provide a **page-table length register (PTLR)** indicating the size of the page table

Memory access times can be sped up using **translation look-aside buffer (TLB)** made up of

- key: used to compare if page table entry is found. If page number is not in TLB, results in **TLB miss** and address translation proceeds as normal

  - TLBs sometimes also store **address-space identifiers (ASIDs)** for each TLB entry that uniquely identifies each process. Used for address-space protection

- value: associated address field returned

Bits can be assigned to each page to specify protection (e.g. read-write or read-only). When the physical address is being computed, the protection bit can be checked to verify protection

- **valid-invalid** bit is usually attached to each entry in the page table

**Reentrant code** can be shared between processes and cannot change during execution, allowing processes to execute the same code at the same time

- Each process has its own copy of registers and data storage to hold data during process execution

## 3.4 Structure of Page Table

### 3.4.1 Hierarchical Paging

Depending on the size of the system, there could be a large number of page tables.

- **forward-mapped**: we can use two-level paging to reduce space overhead so the logical address page number is split into 2 page numbers and maintains the same offset

### 3.4.2 Hashed Page Tables

Uses has value as the virtual page number

- Hash table maintains linked list of elements that has to the same location
- Each element consists of virtual page number, mapped page frame, and pointer to next element in the linked list

Variation of this scheme uses **clustered page tables** where each entry in the hash table refers to several pages, rather than one page

### 3.4.3 Inverted Page Tables

Main issue with page tables is that there might be several entries. Inverted page table has one entry for each real page or frame of memory. Each entry consists of

- Virtual address of page
- Information about the process owning the page

Thus only one page table is in the system with only one entry for each page of physical memory

## 3.5 Swapping

A process, or a portion of a process, can be **swapped** out temporarily out of memory to a **backing store** and then brought back into memory for continued execution

- allows total physical address space of all processes to exceed real total physical memory of system
- increases degree of multiprogramming of system

### 3.5.1 Standard Swapping

Moves entire processes between main memory and backing store (usually a fast secondary storage)

- When a process is swapped to the backing store, any data structures associated with the process must also be written to the backing store
- For multithreaded processes, all per-thread data structures must also be swapped as well
- Idle processes are good candidates for swapping
- Typically slow since we need to move entire processes from memory and backing store

### 3.5.2 Swapping with Paging

Swaps pages of processes, avoids the cost of swapping entire processes. This procedure is usually called **paging**

- We **page out** pages from memory to backing store and **page in** pages into memory

# 4 Virtual Memory

Benefits of Virtual Memory

- User can write programs for extremely large virtual address space
- More programs can be run at the same time, increasing CPU utilization
- Sharing libraries or memory between processes

**Virtual memory** separates logical memory from physical memory

## 4.1 Demand Paging

Pages are loaded only when they are **demanded** during execution

- Need to be able to restart any instruction after **page fault**
- Valid-invalid bit is given to pages to determine if they are legal and in memory
- **page fault** process tires to access page not in memory. Results in a trap that is handled
    1. Check internal tables in PCB to check if reference is valid address
    2. If invalid address, terminate process
    3. Free a frame if necessary
    4. Read desired page into allocated frame
    5. Once read is finished, modify internal table to indicate page is in memory now
    6. Restart instruction that was interrupted by trap

Hardware necessary for pagign and swapping

- Page table with valid-invalid bit
- **Swap space8*: secondary memory that holds pages not in main memory

**Free-frame list**: pool of free frames for satisfying page faults

Major tasks of page-fault service

- Service page-fault interrupt
- Read the page
- Restart the process

## 4.2 Copy-on-Write

For `fork()`, the child often calls `exec()` so copying the parent's address space is unnecssary.

We use **copy-on-write** instead where the parent and child share the same pages that are marked as copy-on-write pages

- If either process writes to a shared page, a copy of the shared page is created

## 4.3 Page Replacement

**Over-allocation** of memory: when a page fault occurs, the system finds the desired page on secondary storage but there are no free frames on the free-frame list

- Can terminate process, standard swapping (large overhead), or **page replacement**

Basic idea of page replacement is to free a frame by writing its contents to swap space and changing the page table to indicate the page is no longer in memory. Page-fault service routine is now

1. Find location of desired page in secondary storage
2. Find a free frame (if no free select a **victim frame** and write it to secondary storage and update page tables)
3. Read desired page into newly freed frame and update page tables
4. Continue process from where page fault occurred

Routine results in 2 page transfers (page-out and page-in). Instead can use a **dirty bit** to examine if a page has been modified when it is being replaced.

Two major implementation problems with demand paging

- **frame-allocation algorithm**: how many frames to gives to each process
- **page-replacement algorithm**: how to select which frames to replace to minimize page-faults

Page-replacement algorithm is tested using a string of memory references (**reference string**) and computing number of page faults

- Only look at the page number
- If immediate references have same page number, ignore them

As number of frames increases, the number of page faults drop

#### 4.3.1 FIFO Page Replacement

Associates each page with the time when it was brought into memory. Oldest page is replaced

- Performance is not always good since we can end up swapping popular pages
- **Belady's anomaly**: page fault may increase when number of allocated frame increases

#### 4.3.2 Optimal Page Replacement

Replaces page that will not be used for the lognest period of time

- Lowest page-fault rate and doesn't suffer from Belady's anomaly
- Difficult to implement since it requries future knowledge of the reference string

#### 4.3.3 LRU page Replacement

Each page associated with the time of the page's last use. LRU chooses page that has not been used for the longest period of time. 2 ways of implementing

- **Counters**: associate each page-table entry with a last used counter (requires looping through pages to find lowest counter)
- **Stack**: keep a stack of page numbers. When a page is referenced, move that page number to the top of the stack. The bottom is the LRU replacement

Doesn't suffer from Belady's anomaly and uses TLB to speed up memory referencing

#### 4.3.4 LRU-Approximation Page Replacement

SKIPPING THIS FOR NOW

#### 4.3.5 Counting-Based Page Replacement

Uses a counter for the number of references that have been made to each page

**Least Frequently Used (LFU)**: replaces page with the smallest reference count. Issue of pages that are heavily used in the beginning getting stuck in the frames

- **Most Frequently Used (MFU)**: idea is that the most recent page won't be used again for awhile

Neither are good

### 4.3.6 Page Buffering Algorithms

Keep a pool of free frames. When a page fault occurs, a victim frame is select, but the desired page is read into the free frame before the victim is written out, minimizing wait time. Once the victim frame is written out, it is added to the free-frame pool

## 4.4 Allocation of Frames

Need to figure out how many frames to allocate for the operating system and for user programs. In general we have

- Operating system must allocate all its buffers and table space from the free-frame list
- When this space is not in use by operating system, it can be used to support user paging
- Leave 3 free frames reserved on the free-frame list to speed up page swapping

### 4.4.1 Minimum Number of Frames

Need enough frames to hold all pages that a single instruction can reference otherwise the instruction can never complete

Minimum is defined by the computer architecture. Maximum is defined by the amount of available physical memory

### 4.4.2 Allocation Algorithms

**Equal Allocation**: split $m$ frames among $n$ processes evenly so each gets $m/n$ frames

**Proportional Allocation**: allocate available memory to each process according to its size. Given $m$ frames, allocate $a_i$ frames to process $p_i$ of size $s_i$

$$a_i = s_i/(S * m)$$

Both algorithms are affected when multiprogramming level is increased and decreased

### 4.4.3 Global and Local Allocation

Since various processes compete for frames, we have 2 types of replacements

**Global Replacement**: processes can select a replacement frame from the set of all frames (even if it is allocated to another process)

- Paging depends on the paging behavior of other processes so paging time can vary

**Local Replacement**: process can only select from its own set of allocated frames

- Paging time is consistent but lowers throughput

Global replacement can be implemented using a strategy where we trigger page replacement when the free-frame list falls below a certain threshold, rather than 0

- When the list size drops below the threshold, a kernel routine (**routines**) begins reclaiming pages from all processes in the system and use a replacement algorithm until the size of the free list reaches the maximum threshold

### 4.4.4 Non-Uniform Memory Access

On **non-uniform memory access (NUMA)** systems, main memory is not created equal so the CPU can only access some sections of it faster than others.

- System has multiple CPUs each with their own local memory

## 4.5 Thrashing

Consider a process that doesn't have enough frames to support all the pages it needs. Here, when a page-fault occurs, it must replace a pages that are in active use, which results in more frequent page-faults

**Thrashing**: high page activity where process is spending more time pagign than executing

### 4.5.1 Cause of Thrashing

Consider a scenario where CPU utilization is low so the operating system increases the degree of multiprogramming and introduces another process.

Global page-replacement is used and pages are replaced regardless of which process they belong to

Now consider if a process enters a active execution phase and needs more frames, causing page faults that affect other processes

This reduces CPU utilization, so the system decides to increase degree of multiprogramming, resulting in even more page faults

To prevent thrashing we need to give each process as many frames as it needs. This can be decided using **locality model** where as a process executes, it moves from locality to locality

- **Locality**: set of pages that are actively used together. A program is composed of several different localities
- Example of locality is when a function is called, all of its memorry access to local variables are in the same locality

Thus the number of frames we should allocate to each process is the number of pages needed for its locality. Otherewise, the process will keep thrashing

### 4.5.2 Working-set Model

Uses a parameter $\Delta$ to define the **working-set window** where we only examine the most recent *Delta* page references (**working set**)

- When a page is active, it will be in the working set
- When a page is no longer being used, it is dropped from the working set $\Delta$ time units after its last reference

Thus working-set model is an approximation of locality

### 4.5.3 Page Fault Frequency

Thrashing results in high page-fault rate so we want to control the page-fault rate

- When it is too high a process needs more frames
- If it is too low, the process has too many frames

### 4.5.4 Current Practice

Current practice is to give computer system enough physical memory when possible to avoid thrashing and swapping

## 4.6 Memory Compression

Instead of paging we can use **memory compression** where we compress several frames into a single frame, reducing memory usage without swapping pages

## 4.7 Allocating Kernel Memory

Free-page list for user processes is maintained by the kernel

Kernel memory is usually allocated from a free-memory pool separate from the free-page list. There are 2 reasons for this

- Kernel requests memory of varying size and must use memory conservatively to minimize waste due to fragmentation
- Certain hardware devices access physical memory directly, without using virtual memory, and may require memory is reside in contiguous pages

### 4.7.1 Buddy System

Memory is allocated as powers of 2. A request not appropriately sized is rounded up to the next power of 2

Example: consider a request of 21 KB of memory and segment is 256 KB

- Divide segment into $A_L$ and $A_R$ of size 128 KB
- Select one $A$ segment and subdivide it into $B_L, B_R$ each of size 64 KB
- Select one $B$ and subdivide it into $C_L, C_R$ each of size 32 KB
- Select one $C$ to satisfy 21 KB request

This system allows **coalescing** to occur easily

- When $C_L$ is released, it can be coalsced with $C_R$ causing $B_L, B_R$ and $A_L, A_R$ to coalesce

However, system can result in internal fragmentation

### 4.7.2 Slab Allocation

**Slab**: one or more physically contiguous pages

**Cache**: consists of one or more slabs

Each kernel data structure has a single cache, which is populated with **objects** that are instantiations of the kernel data structure the kernel represents

Slab-allocation algorithm uses caches to store kernel objects. The slab allocator will attempt to statisfy a request with a free object in a partial slab. Otherwise a free object is assigned from an empty slab

Benefits of slab allocator

- No memory waste due to fragmentation since each unique kernel data structure has an associated cache made up of slabs divided into appropriate sized chunks for the objects
- Memory requests can be satisfied quickly

## 4.8 Other Considerations

Aside from replacement algorthm and allocation policy, there are other considerations

### 4.8.1 Prepaging

Tries to prevent high level of initial paging by bringing some of the pages into memory at one time

- In a working-set model, for each process, we could keep a list of pages in its working set so when the process is suspended, we remember the working set and reload all of them when the process resumes

### 4.8.2 Page Size

Decreasing page sizes increases the number of pages and size of page table but memory is better utilized with smaller pages since it avoids internal fragmentation

### 4.8.3 TLB Reach

Increasing the number of entries in TLB can increase the **hit ratio**. However this is not cheap

Instead we use **TLB reach**: the amount of memory accessible from TLB

- Ideally the working set is stored in TLB otherwise the process will spend a considerable amount of time resolving memory references to the page table

- TLB reach can also be increased by increasing the page size, but this can lead to fragmentation