CMPT 300: Operating Systems I

A Course Overview

Jeffrey Leung

Simon Fraser University

Fall 2018

Contents

1 Introduction

- **Operating system:** An environment or platform for other programs to do work
- Tasks:
  - Allocates and manages usage of all resources
    - Manages conflicting requests for efficiency
    - Important in large-scale multi-user systems
    - May limit performance to improve hardware life
  - Controls execution of programs
    - Prevents errors (e.g. corrupting hardware, the operating system, or other users' files)
    - Prevents undesired manipulation (e.g. buggy or malicious code)
  - Makes the interface easier to use
    - Provides more power to execute programs and actions
    - Important in consumer-facing technology
- **Shared memory:** Common bus which acts as a communication platform between components (e.g. CPU, disk controllers, USB controllers, graphics adapter)
  - CPU asynchronously requests information from a device controller, which writes data to a local shared buffer and sends an interrupt event
- **Interrupt:** Asynchronous event notification created by hardware (memory, I/O, timers, etc.) to interrupt the CPU during normal processing to respond to the event
  - Operating systems are interrupt-driven
  - CPU state is preserved temporarily in registers
  - Contains a specific interrupt number
  - **Interrupt vector:** Information at a known memory location which contains the memory address of an ISR which is mapped to by a specific interrupt number
  - **Interrupt Service Routine (ISR):** Subroutine called to process the completed result alerted by an interrupt
- **Trap/exception:** Synchronous error notification caused by software error
  - E.g. Division by 0, overflow, illegal operation/address

1.1 Privileges

- **Modes:** Level of privilege of the current user (i.e. user and kernel)
  - Hardware mode bit determines the current mode
  - Ensures safe operation
- **Privileged instructions:** Subset of instructions which may cause harm (e.g. I/O control, timer management, interrupt management)
  - Always executed in kernel mode (interrupt to switch system to kernel mode will occur if not already in kernel mode)
- **System call:** Instruction which allows a user to perform privileged instructions which access OS services

- Uses software-generated interrupts
- Associated with a unique number
- Process:
  * The mode is changed to kernel mode
  * Values (e.g. parameters) are checked for validity
  * The system call is executed
  * The mode is changed to user mode
- Usually written in C or C++
- Common APIs (Application Programming Interfaces):
  * POSIX API (Unix, Linux, Mac OS X)
  * Java API (Java Virtual Machine - JVM)
  * Win32 (Windows)
- See figure 1 for a diagram of the system call interface between the application and the OS

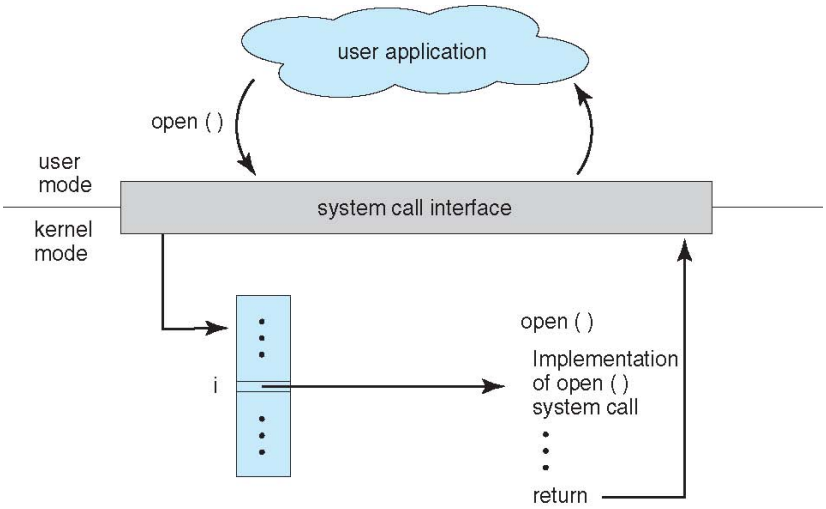

Figure 1: System call interface between the application and the OS

- **Timer chip:** Component in hardware which prevents users from accessing resources for too long
  - Process:
    * OS sets a millisecond timer before providing control to a user program
    * When the period expires, the timer chip sends an interrupt
    * CPU decides whether to grant more time or terminate the program

1.2 Structure of an OS

- Startup/boot:
  - OS must be available to hardware when starting
  - Process:

    * Instruction register is loaded with a predefined memory location which is the address of the bootstrap loader in ROM
    * Bootstrap loader performs diagnostic tests (Power-on Self Testing)
    * From a fixed file location on the disk (the boot block), code is loaded into memory
    * Remainder of the loader is loaded
    * Kernel is loaded

- **Simple/monolithic structure:** OS compositional design where the entire system is a single component with no access controls
  - Used by UNIX in the past
  - Pro: Efficient if constructed well
  - Con: Difficult to implement and maintain
  - For an example diagram, see figure 2

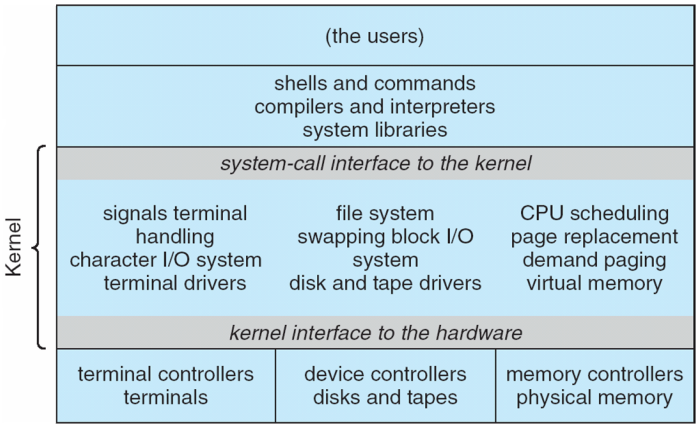

Figure 2: Monolithic structure

- **Layered structure:** OS compositional design where the system is divided into layers, each of which only communicates with the layer directly beneath it
  - Pro: More reliable than a monolithic structure due to better modularization
  - Con: Less performant than a monolithic structure due to more switching between handlers
  - For an example diagram, see figure 3
- **Microkernel structure:** OS compositional design where user programs only communicate with hardware through the kernel when necessary
  - **Interprocess communication:** Kernel passes messages between modules
  - Pros: Easier to extend and port to new architectures, more secure due to services being isolated to user mode
  - Cons: Less performant due to indirect communication
  - For an example diagram, see figure 4
- **Modular structure:** OS compositional design where the system is comprised of independent components

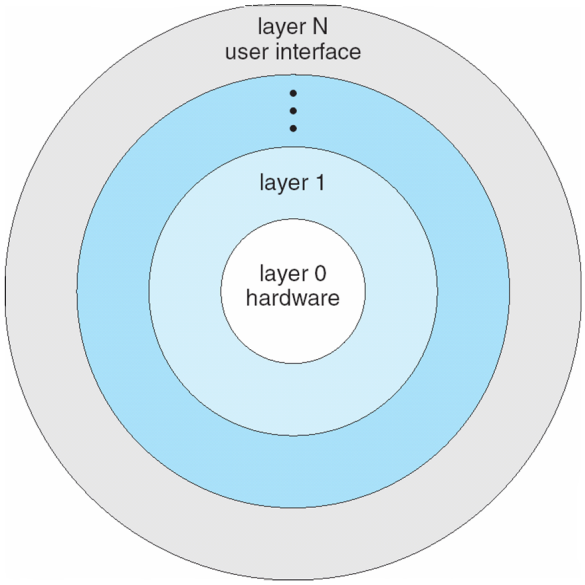

Figure 3: Layered structure

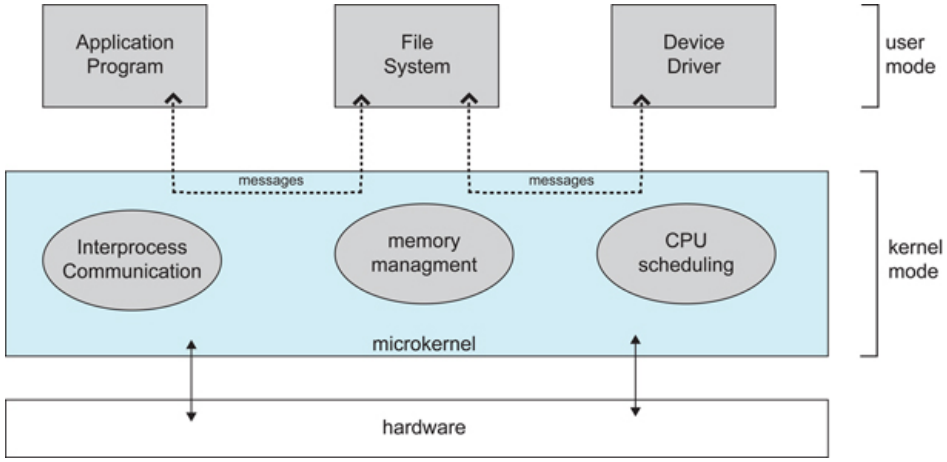

Figure 4: Microkernel structure

- Kernel module contains core components; other modules are linked to the kernel module as needed
- Pros:
  - No need to recompile all code, only the modules with changes
  - Modules communicate using defined interfaces
  - Easy to maintain, update, and debug
  - More flexible than layers
  - More efficient than microkernels due to direct communication between modules
- Used by modern OSes such as Solaris, Linux
- For an example diagram, see figure 5

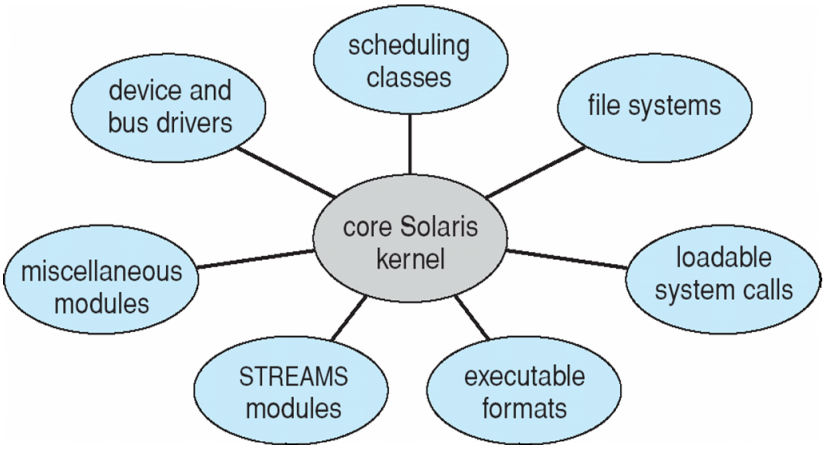

Figure 5: Modular structure

- **Hybrid structure:** OS compositional design where the system uses multiple design philosophies
  - Used by many OSes such as Mac OS X (e.g. has layers and a microkernel, the kernel environment is monolithic, and lower-level drivers are modules)
  - For an example diagram, see figure 6

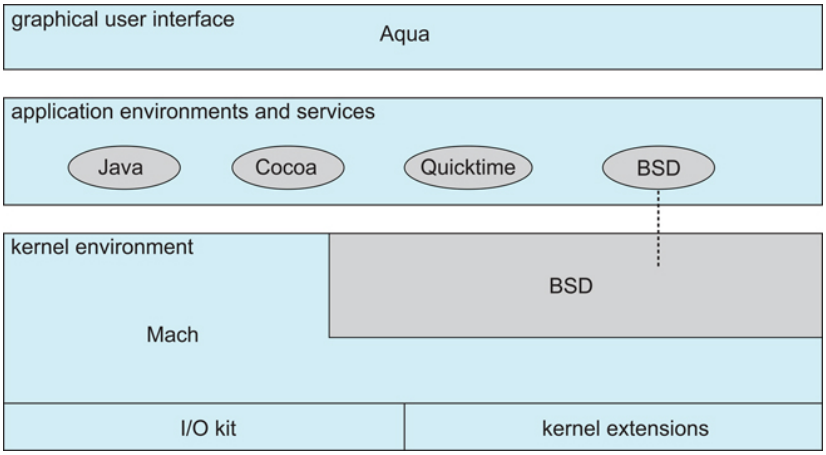

Figure 6: Hybrid structure

2 Processes

- **Process:** Program which is loaded in memory and currently being executed (i.e. the required OS structures exist)
  - Consists of a program counter, stack (which includes local variables and return address), and data location/area (which includes global variables and dynamically allocated memory) (see figure 7)

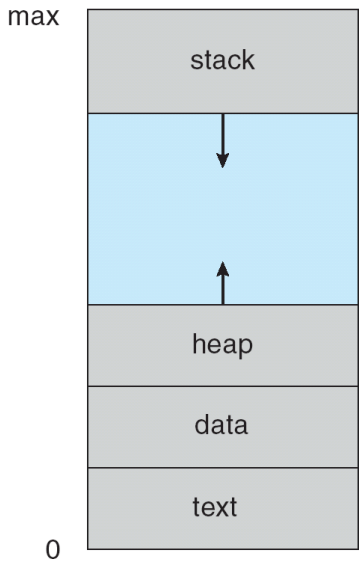

Figure 7: Process Data

  - Created by a parent process; may create children processes
- **Process state:** Current execution stage of a process
  - New, running, waiting (for an event), ready (waiting for CPU), terminated
- **Process Control Block (PCB):** Set of data about a particular process
  - Used to switch the CPU between processes
  - Data includes process state, program counter, CPU registers, CPU scheduling information, memory management information, I/O status information
- Managed in scheduling queues
  - **Ready queue:** Set of processes in memory which are ready to execute
    - **Device queue:** Set of processes in memory which are waiting for an I/O device
  - See figure 8
- **Context switch:** CPU changing from one process to another
  - State of the first process is saved in a PCB in memory; state of the second process from another PCB in memory is loaded into the registers
  - **CPU state:** Values of the CPU registers
    - Includes program counter and stack counter
  - Overhead with no work done

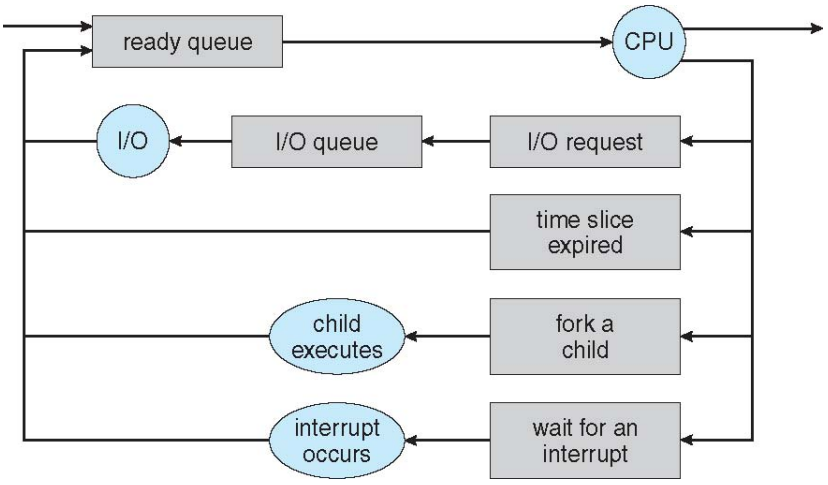

Figure 8: Scheduling Queues

- Usually several milliseconds
- Some systems use multiple register sets which can be easily switched between using pointers
- **Fork:** System call executed by a process (parent) which creates a new process (child)
  - Child is a copy of the parent
  - Child loads another program using the `exec()` system call
  - Parent waits for children to exit, or aborts the execution of its children
    - **Cascade termination:** Abortion of the children of a child process
    - OS deallocates resources
    - Data is returned to parent via `wait()`
    - **Zombie process:** Process which has been terminated but whose parent has not yet called `wait()`
      - Return value held temporarily in memory
    - **Orphan process:** Process which is running but its parent has exited (without calling `wait()`)
      - **Adoption:** Changing the parent of an orphaned process to the `init` process

2.1 Threads

- **Thread:** Sequence of instructions which the CPU can execute as a unit
  - Tracked by program counter, register set, and stack
  - Created by system call `clone()`
    - Can be parameterized to share or duplicate resources
  - Equivalent to processes in Linux
- **Multithreading:** Utilization of multiple threads by a single process
  - Threads of the same process share a code section, data section, and OS resources

  - Benefits: Responsive, shares resources, scalable
  - If `fork()` is called by a process with multiple threads, all threads are duplicated unless `exec()` is called immediately afterwards
- **User-level thread:** Thread operating only in user space
  - Data structures and thread operations are in user level/mode only
  - Kernel is not aware of user-level threads
- **Kernel-level thread:** Thread operating only in kernel space
  - Data strcutures and thread operations are in kernel level/mode (requiring system calls)
  - Provided by the kernel
- Mappings from user threads to kernel threads:
  - One-to-one: User level threads are mapped individually to kernel threads
    - Pros: Increased concurrency (multiprocessors can be used); one thread cannot block another
    - Con: Significant overhead for thread resource usage and management
  - Many-to-one: Threads are managed in user level to have several user threads map to a single kernel thread
    - Pro: Faster thread management due to no switching or system calls
    - Cons: A single thread can block all other threads; incompatible with multiprocessors
    - E.g. Solaris Green Threads library
  - Many-to-many: Multiple user threads are mapped to multiple kernel threads
    - **Thread scheduler:** User-level module which manages mapping of user threads to kernel threads
      - **Upcall:** Notification by kernel when a user thread blocks so another user thread can use the free kernel thread
        - Complex processing to redirect other threads, nearly duplicating kernel-level work
      - Pros: Increased concurrency and flexibility
      - Con: Difficult to implement
- E.g. POSIX Pthreads API specification used in UNIX systems
  - `pthread_create()` blocks until `pthread_exit()`
  - `pthread_join()` combines the two threads
- **Thread pool:** Limited set of user-level threads which are either available for use or in use
  - Removes the overhead of creating new threads
  - When a new request is created, an inactive thread is pulled from the pool to service the task
  - Used commonly in web servers to service incoming requests

2.2 Signals

- **Signal:** Event notification sent to a process
  - **Synchronous signal:** Signal to the same process which caused the signal
    - Specific to the thread which caused the signal

    - E.g. Illegal memory access
  - **Asynchronous signal:** Signal from one process to another
    - Delivered to all threads of the process
    - E.g. Cancelling a running process with Ctrl–C
- **Signal handler:** OS or user-defined module which processes a signal
- Comparison to interrupts (see figure 9):

Figure 9: Comparison between Interrupts and Signals

	Interrupts	Signals
Initiated by	CPU (e.g. division by 0), I/O devices (e.g. user input), or software (e.g.system call)	Kernel (e.g. SIGIO) or process (e.g. SIGKILL)
Handled by	Interrupt Service Routine in kernel space	Kernel (default) or process (manual override) in user space
Continuation	May map to signals and deliver to processes for specific handling	May have complex logic to call other functions in kernel

3 Scheduling

- Process execution switches between bursts of CPU execution and I/O wait
  - **I/O bound process:** Process which spends more time in I/O waits than CPU execution
  - **CPU bound process:** Process which spends more time in CPU execution than I/O waits
-
- **Long-term/job scheduler:** Component which decides which processes should be moved to the ready queue, to maintain a balanced mix of I/O-bound and CPU-bound processes
  - Controls degree of concurrency/multiprogramming
  - Slow; invoked infrequently
- **Midterm scheduler:** Component which holds partially-executed processes which are not in the ready queue
- **Short-term/CPU scheduler:** Component which decides which processes from the ready queue the CPU should perform work for
  - Quick; invoked frequently (every several milliseconds)
  - **Nonpreemptive:** Process which can only stop during CPU execution by terminating or by waiting (e.g. I/O request, child termination)
  - **Preemptive:** Process which can be forcefully stopped (preempted) during CPU execution by the OS (e.g. to run a higher-level process)
    - More difficult to implement
      - Needs to maintain consistency of data shared between processes and kernel data structures (e.g. I/O queues)
      - Needs hardware support (e.g. timers)
- **Dispatcher:** Component which allocates the CPU to various processes by switching context, switching user mode, jumping to the specified location in the process, and restarting execution
  - **Dispatch latency:** Time required for the dispatcher to switch tasks

3.1 Scheduling Algorithms

- Criteria:
  - Maximize CPU utilization
  - Maximize throughput
  - Minimize first response time
  - Minimize waiting time (time before a process starts exeuction)
  - Minimize job turnaround time
- **Convoy effect:** Suboptimal situation where a long process blocks the CPU, requiring shorter processes to wait
  - Utilization of CPU and device is low
- **First-Come, First-Served (FCFS):** CPU scheduling algorithm which prioritizes processes based on when they are placed in the queue

- Tasks which take longer will block other tasks; a CPU-bound process would block I/O-bound processes
- Examples:
  - For tasks $P_1 = 24$ms, $P_2 = 3$ms, $P_3 = 3$ms in the given order (see diagram 10), the average waiting time is $\frac{0+24+27}{3} = 17$ms

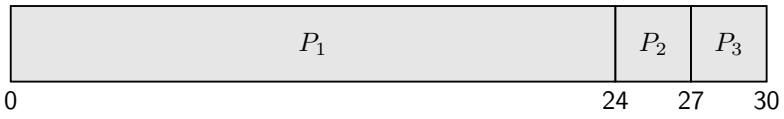

Figure 10: FCFS Gantt Chart for $P_1 = 24$ms, $P_2 = 3$ms, $P_3 = 3$ms

  - For tasks $P_2 = 3$ms, $P_3 = 3$ms, $P_1 = 24$ms in the given order, the average waiting time is $\frac{0+3+6}{3} = 3$ms

- **Shortest Job First (SJF):** CPU scheduling algorithm which prioritizes processes based on shortest execution time (without preempting currently running processes)
  - Provably optimal in giving minimum average waiting time for a given set of processes
  - Length of next CPU burst is available for long-term scheduling but not available for short-term scheduling
  - Time of execution is approximated through prediction from previous executions
  - Examples:
    - For tasks:
    - Time prediction
- **Shortest Remaining Time First (SRTF):** CPU scheduling algorithm which prioritizes processes based on the least execution time remaining, switching between processes when necessary
  - Examples:
- **Round Robin:** CPU scheduling algorithm which provides a unit of time (q)) for each process before moving to the next process
  - **Time quantum** (q): Unit of CPU time (10-100 ms) for which a process can execute
    - For n processes in the ready queue, each process receives $\frac{1}{n}$ of CPU time
    - q and context switch time
  - Process is moved to end of (circular) ready queue once its time is finished
  - Better response time and higher average turnaround time (vs. SJF)
- **Priority scheduling:** CPU process scheduling which allocates work depending on a priority level
  - SJF is when the priority is the inverse of the CPU burst length
  - **Starvation:** Failure to allocate a low-priority process
  - **Aging:**
- **Multilevel queue scheduling:** Scheduling design with multiple prioritized ready queues, each with its own scheduling algorithm
  -
- **Multiprocessor scheduling:**

- Divide load among multiple processors
- **Asymmetric multiprocessor:** Multiprocessor scheduling strategy where one master processor handles all scheduling
- **Symmetric multiprocessor:** Multiprocessor scheduling strategy where
  - Issues:
    - **Processor affinity:** Binding and unbinding of a process to the CPU
      - When a process is migrated to a new processor, the old cache must be invalidated and the new cache must be re-populated which has a performance penalty
    - **Load balancing:** Workload distribution among various processors
      - **Push migration:** Task repeatedly checks all processors and moves tasks to evenly distribute them
      - **Pull migration:** Idle processor taking a task from a busy processor
    - Tradeoff between processor affinity and load balancing

- **Real-time scheduling:**
- **Completely Fair Scheduler:**
  - **Target latency:** Duration of time during which every runnable task should be executed at least once
    - **Nice value:** Duration of time during which...
- How algorithms are designed:
  - Use deterministic modelling to run algorithms on workloads
  - Use queuing models and theory to analyze algorithms (using typically unrealistic assumptions)
  - Use a simulator to model systems (using a synthetic workload or traces from real systems)
    - Expensive and time-consuming

4 Inter-Process Communication

- **Inter-process communication (IPC):** Sharing of information from one process to another
  - Purposes:
    - Faster computation
    - Increased modularity
    - Increased convenience
- **Shared memory:** Method of IPC in which a single memory space is used by multiple processes
  - Processes attach the shared memory space created by another process to their own address space
  - Pros: Fast, convenient
  - Cons: Requires synchronization to prevent conflicts
  - Implementations (POSIX): `shm_open()` to create a shared memory space and `mmap()` to create a mapping from a process to the shared space
- **Message passing:** Method of IPC in which a process sends a message to another process through the kernel
  - Can be to a process directly or can use constructs such as ports, mailboxes
  - Process can block until a response is received
  - Unsent messages may buffer and be placed in a waiting queue
  - Diagram: See figure 11
  - Pros: No conflicts possible
  - Cons: Slow, requires overhead through system calls and kernel

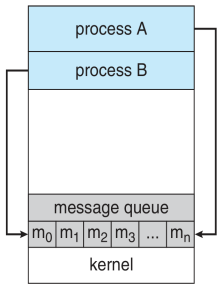

Figure 11: Diagram of message passing

5 Synchronization

- **Race condition:** Data inconsistency when multiple processes manipulate shared data concurrently and the result varies depending on the order of execution
  - To avoid, set code as the **critical section** so only one process can execute their CS at the same time
- **Producer-Consumer Problem:** Two processes (threads) share a buffer where the producer thread places items into the buffer (waiting if full) and the consumer thread removes items from the buffer (waiting if empty)
  - Both threads should update a counter of the number of items in the buffer
    - *Issue:* Race condition may occur when the counter is incremented or decremented, if the CPU scheduler switches between them at that instruction
  - *Issue:*
- **Semaphore:**
  - Non-busy waiting semaphore: Integer value with queue
    - Can block to suspend the process or
  - *wait* and *signal* are critical sections to avoid further race conditions
    - Busy waiting is shortened
  - Value of 0 (false) means that a process is waiting; value of 1 (true) means that no process is waiting
  - Issue: Process may release lock and then immediately relock, which creates a problem of prioritization
  - If a part of a process P2 should be executed only after the execution of a part of another process P1, then there can be a *signal* after the part in P1 and a *wait* before the part in P2
  - Issue: Deadlock can occur if multiple sempahores are accessed in the incorrect order, if processes block forever and do not release resources,
    - Example: When both process wait for signals from the other process, which occur after the waits
    - E.g.
- Classical synchronization problems: Abstractions which can...
  - **Bounded-Buffer (Producer-Consumer) Problem:** See above
    - Issues are a violation of modification of the count, producing when full, and consuming when empty
      - Solution: Use 3 semaphores - mutex initialized to 1, full initialized to 0, empty initialized to N
  - **Readers-Writers Problem:**
  - **Dining-Philosophers Problem:** 5 philosphers in a circle have one chopstick in between each of them
    - Solution: Use 5 semaphores (chopstick [0..5])
      - Deadlock: If every thread picks up one chopstick, then no chopsticks are available to unblock

        - Deadlock solution: Take both chopsticks only when available
        - Deadlock solution: Odd threads will take only one side first and even threads will take only right side first

5.1 Deadlocks

- **Deadlock prevention:**
- **Deadlock avoidance:**
- **Deadlock detection:**
  - Wait-for graph
  - Algorithm invoked when CPU utilization is less than 40% (occurs occasionally)
- **Deadlock recovery:** Removal of a deadlock situation
  - **Victim:** Process which is targeted in an attempt to remove a deadlock
    * Determined by priority, execution time, remaining execution time, resources used, resources needed, number of processes required to be terminated, interactive/batch execution
  - *Process termination:* A process is aborted continually until the deadlock cycle is eliminated
    * At least one process for each disjoint cycle must be aborted
  - *Resource pre-emption:* A resource is pre-empted from some processes
    * **Rollback:** Returning of a process and its resources to a safe state and restarting it
    * *Starvation:* Same process may be continually re-chosen as victim
- **Memory Management Unit (MMU):** Hardware component which translates logical addresses to physical addresses
  - Hardware, not software because of high usage (every CPU instruction requires the MMU)
  - Memory protection:
    * If an address out of bounds is accessed, the MMU will raise a trap which is translated to a segmentation fault
- Memory has two partitions for resident OS and user processes
- **Contiguous allocation:**
  - Requires management of information about allocated and free partitions
  - **Memory fragmentation:** Dissociation of usable memory holes across a large space
    * **External fragmentation:** Situation where there exists enough memory to satisfy a request but the memory is fragmented across memory and not contiguous
    * **Memory compaction:** Moving contents of memory to combine free memory into a larger block
      - Only possible if relocation is dynamic during execution time
  - **Internal fragmentation:** Situation in paged memory where the allocated memory is larger than the requested amount, and cannot be utilized as it is internal to the allocated partition
- **Paging/Non-contiguous allocation:** Allocation of memory for a process wherever available
  - **Frame:** Fixed-size block of physical memory containing part of the data of a process, with the same size as a page

    - Sized by power of 2
    - OS tracks free frames
  - **Page:** Fixed-size block of logical memory containing part of the data of a process, with the same size as a frame
  - **Page table:** Translation map which maps logical addresses to physical addresses
  - Address transition:
  - **Hierarchical page table:** Page table which provides address mapping to a set of page tables which maps to process data
    - Pros:
      - Process data page tables do not need to be in contiguous memory
      - Page tables can be created on-demand
    - Cons:
  - **Hashed page table:** Page table which hashes a page number to a page table
    - Collisions create linked list chain of elements which are searched for the correct value
      - Cost of searching is linear
    - Common for address spaces with less than 32 bits
  - **Inverted page table:** Page table which contains an entry for each physical memory frame
    - Table is searched for the matching page number stored in an entry, and moves to the corresponding frame
    - Pros:
      - Only requires one instance for all processes
      - Less memory required to store page table
    - Cons:
      - Increases search time
        - Can be alleviated with a hash table

5.2 Segmentation

labelsubsec:synchronization:segmentation

- **Segment:** Component of a program which is a logical unit
  - Example: Main program, functions/procedures/methods, objects/variables, stack, arrays
- Logical address:
  - Denoted by (*segment number*, *offset*)
- **Segment table:** Table which maps a logical address to a physical address
  - **Base:**
  - **Limit:**
  - Processing steps:
    - Add the segmentation value to the difference Value
    - Check whether the combined value is greater than the limit

* If it is beyond the limit, a trap is raised
* If it is valid, the data is retrieved from the segment

6 Deadlocks

- **Resource:** Limited-quantity component which is needed by a process (e.g. CPU time, memory space, I/O device access)
  - Processes request (and block if resource is not free), use, and release a resource
- **Deadlock:** Situation where a set of blocked processes hold a resource and each needs to acquire a resource held by another process in the set
  - Fundamental conditions:
    - **Mutual exclusion:** Natural condition where a single resource can be held by no more than one process
    - **Hold and wait:** Condition where a process holds a resource and waits to acquire another
    - **No preemption:** Condition where a process cannot be preempted during execution to release its resources
    - **Circular wait:** Condition where there exists a set of processes $P_0, P_1, \ldots, P_n$ such that P_m is waiting for a resource held by P_{m+1} where $m \neq n$, and P_n is waiting for a resource held by P_0
- Resource Allocation Graph (RAG)
- Safety condition: There exists a sequence of all processes $P_1, P_2, \ldots, P_n$ such that each $P_i, 1 \leq i \leq n$, the resources that P_i will request can be satisfied by the currently available resources in addition to the resources held by all $P_j, 1 \leq j < i$.
- **Deadlock prevention:** Strategies to prevent deadlocks through disallowing a specific process behaviour to disable one of the deadlock conditions
  - Preventing hold and wait: A process may only hold one resource at a time (results in low resource utilization)
  - Preventing hold and wait: A process is allocated all its resources before execution (results in starvation)
  - Allowing preemption: A process may be preempted to lose its resources if the next resource request will not be satisfied, or if another process requires its resources (results in failure to maintain atomicity of operations due to resources which cannot easily save states)
  - Preventing circular wait: All resource types have a total ordering and enforce process requests in a relative order
- **Deadlock avoidance:** Strategy to prevent deadlocks through forcing a process to wait if its resource allocation will create a deadlock
- **Deadlock detection and recovery:** Strategy to detect the occurrence of a deadlock in the system and eliminate processes until the deadlock is removed

7 Memory Management

- Code location can be absolute if memory location is known during compile time, or relocatable otherwise
- **Physical address:** Location in physical memory accessed by the memory unit
- **Logical/virtual address:** CPU-managed location in memory
  - Different from physical address if addresses are bound during execution
  - **Memory Management Unit (MMU):** Hardware component which maps virtual addresses to physical addresses during runtime
    - Ensures separation and protection of memory areas
    - Basic components: Base and limit register

7.1 Contiguous Allocation

- Allocation algorithms:
  - **First-fit algorithm:** Allocating memory in the first available sufficiently-sized hole
  - **Best-fit algorithm:** Allocating memory in the smallest sufficiently-sized hole to create the smallest possible remaining hole
    - Requires searching the entire memory space
  - **Worst-fit algorithm:** Allocating memory in the largest sufficiently-sized hole to create the largest possible remaining hole
    - Requires searching the entire memory space
- **External fragmentation:** Situation where the memory space exists to fulfill a request but it is not contiguous
  - **Memory compaction:** Method to solve external fragmentation which places all free memory together
    - Only possible if addresses are bound during execution

7.2 Paging

- **Page:** Fixed-size, contiguous logical memory unit for non-contiguous memory allocation
  - **Frame:** Fixed-size, contiguous physical memory unit for non-contiguous memory allocation
    - Associated 1:1 with each page
- Components of addresses:
  - **Page number:** Component which contains the base address of a page in physical memory
  - **Page offset:** Component which contains an offset of the base address sent to the MMU
  - For m, n where the address space is 2^m and the page size has 2^n entries, the number of pages is $\frac{2^m}{2^n} = 2^{m-n}$, the address has m bits, the page number has $m-n$ bits, and the page offset has n bits
  - Size: Power of 2 (so that the page number and offset can be easily navigated)
- **Page table:** Set of translations from logical memory to physical memory
  - Allows partitioning of physical memory and linkage with virtual memory in arbitrary ordering
  - Specific to each process

– Located through the page table base register

– Requires two memory accesses (one for page table, one for instruction)

– Includes protection bits to specify if a page is read-only/read-write/execute-only, or if a page is valid/invalid access for the process

– Diagram: See figure 12

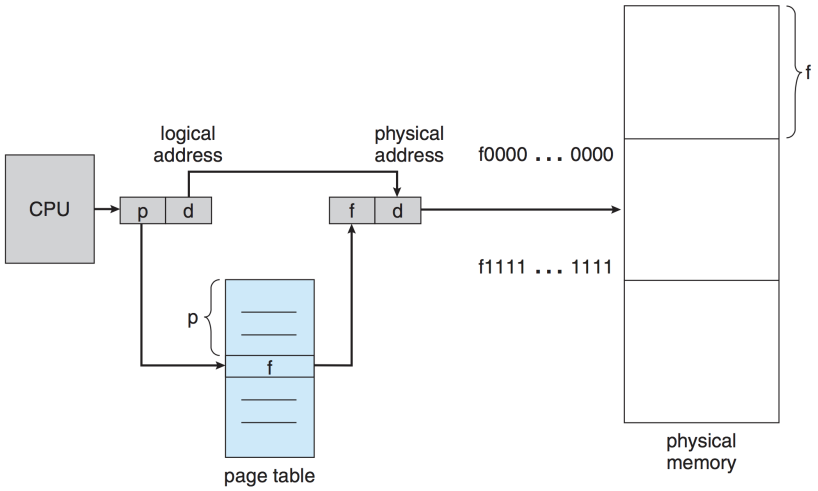

Figure 12: Diagram of a Page Table

– Page table address translation process:

$$
\begin{aligned}
\text{Let } s &= \text{ page/frame size} \\
\text{Let } p &= \text{ page number} \\
\text{Let } d &= \text{ offset}, 0 \leq d < s \\
\text{Logical address } &= p \times s + d \\
\text{Let } f &= \text{ frame number translated from } p \text{ using page table} \\
\text{Physical address } &= f \times s + d
\end{aligned}
$$

  * Example:

$$
\begin{aligned}
\text{Page/frame size } s &= 4 \text{ bytes} \\
\text{Page table: } 0 &\to 5 \\
1 &\to 6 \\
2 &\to 1 \\
3 &\to 2 \\
\text{Logical address } &= 0 \\
&= p \times s + d \\
&= 0 \times 4 + 0 \\
f_{p=0} &= 5 \\
\text{Physical address } &= f \times s + d \\
&= 5 \times 4 + 0 \\
&= 20
\end{aligned}
$$

    * Example:

$$
\begin{aligned}
\text{Page/frame size } s &= 4 \text{ bytes} \\
\text{Page table: } 0 &\to 5 \\
1 &\to 6 \\
2 &\to 1 \\
3 &\to 2 \\
\text{Logical address } &= 13 \\
&= p \times s + d \\
&= 3 \times 4 + 1 \\
f_{p=3} &= 2 \\
\text{Physical address } &= f \times s + d \\
&= 2 \times 4 + 1 \\
&= 9
\end{aligned}
$$

- **Translation Look-aside Buffer (TLB):** Fast-lookup associative memory structure which stores recently used page table entries
    - **Address Space Identifier:** Component of a TLB entry containing a process ID for protection
    - Common across processes
    - If the page is not found, the page table is used and the entry is added to the TLB
        * Some TLB entries are fixed and cannot be replaced (e.g. kernel codes)
    - Diagram: See figure 13
- **Effective Access Time:** Amortized time for TLB lookup and page table access
    - TLB hit ratio is proportional to the effective access time between the time for 1 memory access and the time for 2 memory accesses
    - Equation:

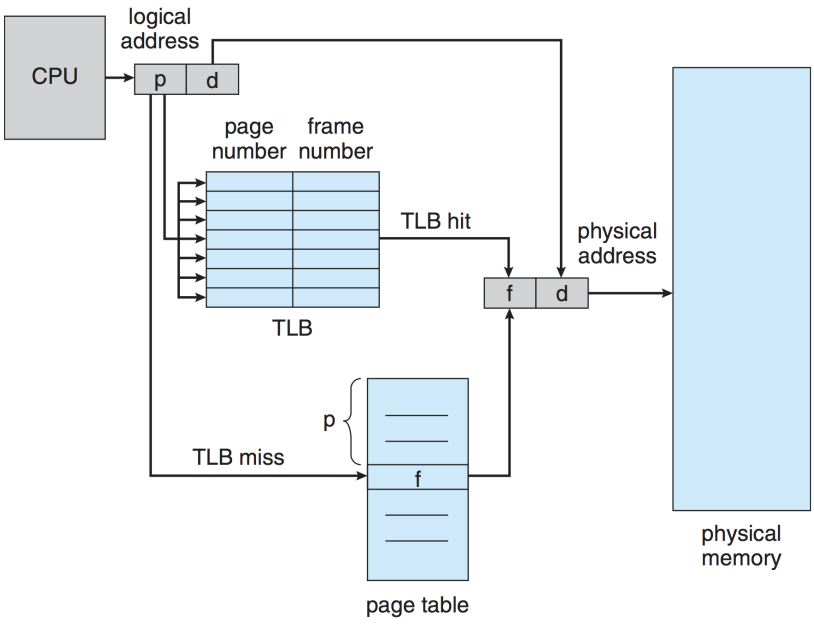

Figure 13: Diagram of a TLB

$$
\begin{aligned}
\text{Let } t_m &= \text{ memory access time} \\
\text{Let } \varepsilon &= \text{ TLB lookup time (significantly less than } t_m) \\
\text{Let } \alpha &= \text{ TLB hit ratio (percentage of times page found in TLB)} \\
\text{Effective Access Time } &= \text{ Time for TLB hit } \times \text{ TLB hit percentage } + \\
&\quad \text{Time for TLB miss } \times \text{ TLB miss percentage} \\
&= (\varepsilon + t_m) \times \alpha + (\varepsilon + 2 \times t_m) \times (1 - \alpha) \\
&= \varepsilon + t_m \times (2 - \alpha)
\end{aligned}
$$

* Example:

$$
\begin{aligned}
\text{Let } t_m &= 100 \text{ ns} \\
\text{Let } \varepsilon &= 0 \\
\text{Let } \alpha &= 0.8 \\
\text{Effective Access Time } &= \varepsilon + t_m \times (2 - \alpha) \\
&= 100 \text{ ns } \times (2 - 0.8) \\
&= 120 \text{ ns} \\
\text{Slowdown } &= \frac{100 \text{ ns}}{120 \text{ ns}} - 1 = 20\%
\end{aligned}
$$

* Example:

$$
\begin{aligned}
\text{Let } t_m &= 100 \text{ ns} \\
\text{Let } \varepsilon &= 0 \\
\text{Let } \alpha &= 0.99 \\
\text{Effective Access Time } &= \varepsilon + t_m \times (2 - \alpha) \\
&= 100 \text{ ns } \times (2 - 0.99) \\
&= 101 \text{ ns} \\
\text{Slowdown } &= \frac{100 \text{ ns}}{101 \text{ ns}} - 1 = 1\%
\end{aligned}
$$

- **Reentrant code:** Read-only code shared among multiple processes
  - Must appear in the same location in the logical address spaces due to processes sharing a TLB
- Page table structural alternatives:
  - Page table requires contiguous memory or the page table will need to be paged itself
  - For an address space of 32 bits and a page size of 4 KB (2^{12} bytes), a process could have $2^{32-12} = 2^{20}$ pages; for a page table with entries of 4 bytes, the size of a page table for each process could be $2^{20} \times 4 = 2^{22}$ bytes $= 4$ MB which is very large
  - **Hierarchical page table:** Method to make page tables less contiguous in memory by paging a page table through partitioning the page table
    - Requires additional memory accesses
    - Diagram: See figure 14
    - Diagram of access: See figure 15
    - Diagram: See figure 14
  - **Hashed page table:** Method to store page table entries non-contiguously by hashing page number to a linked list of elements, each with a page number and associated frame number
    - Diagram: See figure 16
  - **Inverted page table:** Method to use only one page table for all processes by using a table of page numbers (limited by amount of frames) and the process which owns the frame
    - Increases time to find a page reference
    - Creates difficulty in implementing a shared page
- **Internal fragmentation:** Situation where memory allocated as a fixed-size partition is larger than required which creates unusable excess memory

7.3 Segmentation

- **Segment:** Component of a program (e.g. function, object, main program)
  - Segments are paged to avoid external fragmentation so the user interacts with the segments and the OS interacts with the pages
- **Segment table:** Structure which maps a logical address to a physical address using a base address and limit

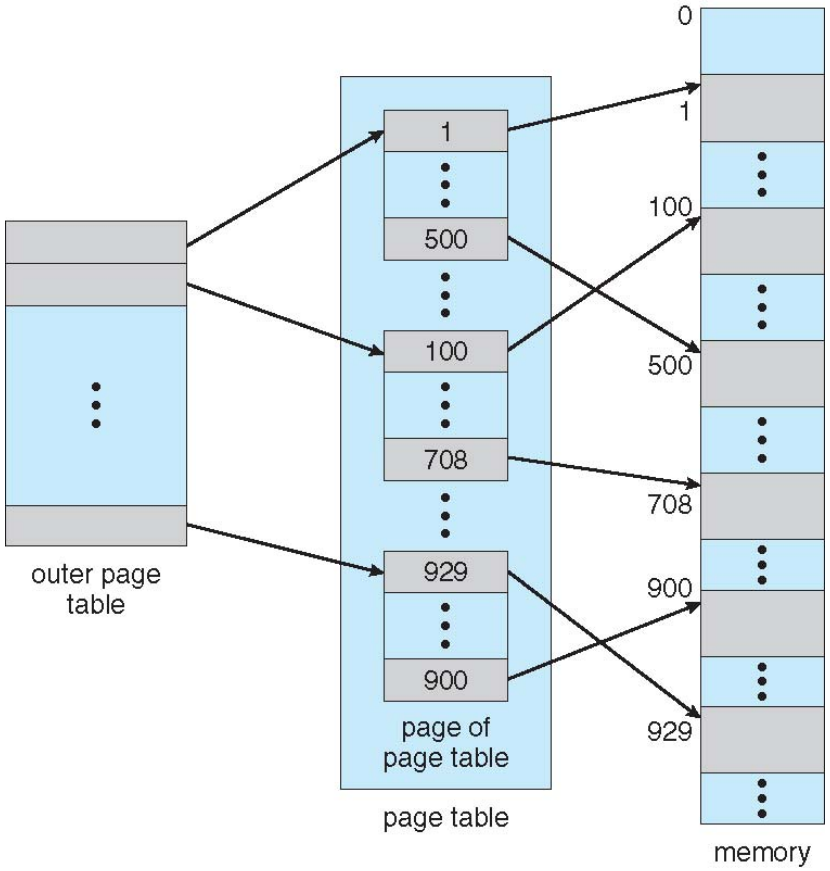

Figure 14: Diagram of a Hierarchical Page Table

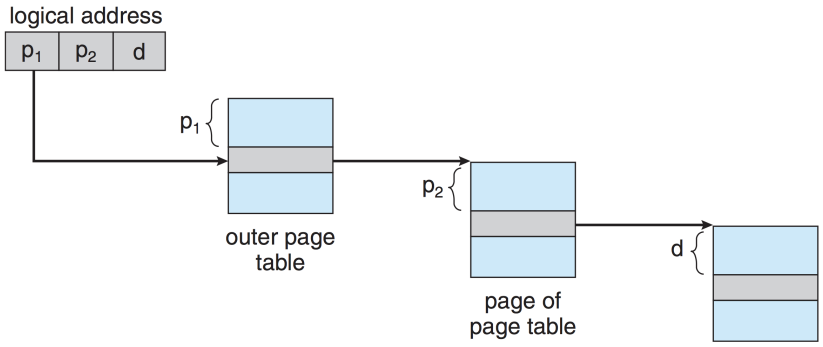

Figure 15: Accessing a Hierarchical Page Table

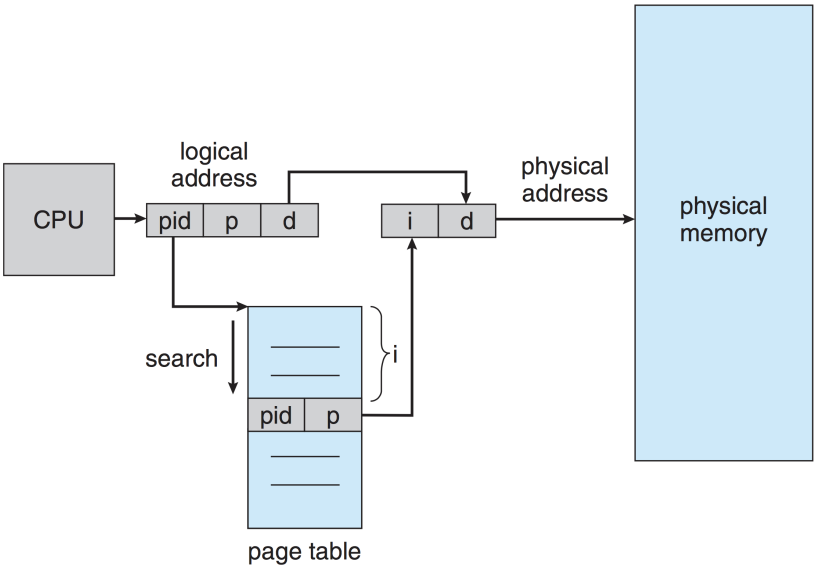

Figure 16: Diagram of an Inverted Page Table

8 Virtual Memory

- **Virtual memory:** Abstraction of memory management where logical memory can be conceptually larger than the physical address space
  - Only part of a program is in memory at any given time
- **Demand paging:** Making pages available in virtual memory only as needed
  - Decreases I/O requirements and waiting time, and increases number of potential concurrent processes
  - Indicated by valid-invalid bit in page table (if invalid, the memory access is either illegal or not yet in memory)

8.1 Page Faults

- **Page fault:** Legal memory access of a page which is not currently in memory which triggers retrieval
  - **Major page fault:** Page is brought from disk
  - **Minor page fault:** Page table is updated (e.g. linking shared pages)
  - If no free pages are available, a page will be removed from memory to make room
  - To preserve atomicity of instructions which modify data, previous register values are saved and hardware accesses both ends of the data before execution to trigger page faults
  - Effective Access Time:

$$\begin{aligned}
\text{Let } p = &\text{ page fault rate, } 0 \leq p \leq 1 \\
&\text{where 0 is no faults, 1 is every reference is a fault} \\
\text{Let } t_m = &\text{ memory access time} \\
\text{Let } t_p = &\text{ page fault time} \\
&\text{(including interrupt, page read, restart)} \\
\text{Effective access time } &= p \times t_p + (1-p) \times t_m
\end{aligned}$$

    - Example:

$$\begin{aligned}
p &= 0.001 \text{ 1 fault per 1,000 memory accesses} \\
t_m &= 200 \text{ ns} \\
t_p &= 8 \text{ ms} = 8,000,000 \text{ ns} \\
\text{Effective access time } &= p \times t_p + (1-p) \times t_m \\
&= 0.001 \times 8,000,000 + (1 - 0.001) \times 200 \\
&= 8,000 + 199.8 \\
&= 8,200 \text{ ns} \\
\text{Slowdown } &= \frac{8200 \text{ ns}}{200 \text{ ns}} = 40 \text{ times slower}
\end{aligned}$$

  - Process:

        * Send a trap to the OS
        * Save the user registers and process states
        * Determine that the interrupt was a page fault
        * Check legality of the page access and determine the location of the page
        * Issue a read from the disk to a free frame
        * Wait until the read request is serviced, with seek and/or latency time
        * Transfer the page to a free frame
        * While waiting, allocate the CPU to some other process
        * Receive a completion interrupt from the disk I/O subsystem
        * Save the registers and process state for the other process
        * Determine that the interrupt was from the disk
        * Correct the page tables to show page is now in memory
        * Wait for the CPU to be allocated to this process again
        * Restore the user registers, process state, and new page table
        * Resume the interrupted instruction

- **Copy-on-Write (COW):** Method of quickly/efficiently creating a child process by sharing the pages of the parent process in memory and copying the data only when a shared page is modified
- **Pre-paging:** Bringing pages into memory before they are referenced to decrease page faults
    - May waste memory and I/O if the page is not used

8.2 Page Replacement

- **Page replacement algorithm:** Method of selecting a victim page to be removed from memory which is designed to minimize fault rate
    - Dirty/modified bit in page table entries to save swap-out overhead if the victim page was not modified
    - **I/O interlock:** Page currently used for I/O is locked in memory and cannot be replaced during a fault
- **First-In-First-Out (FIFO) page replacement algorithm:** Removing from memory the oldest page upon a fault
    - Could consistently remove a heavily-used page
    - **BÃłÃądy's anomaly:** Phenomenon in the FIFO page replacement algorithm where increasing available frames may potentially result in greater page faults
- **Least Recently Used (LRU) page replacement algorithm:** Removing from memory the page which has not been used for the longest period, upon a page fault
    - Counter tracking: Using a data field to store CPU clock of the last memory access and searching for the oldest page
        * Search is time-consuming, many memory writes, potential clock overflow, requires hardware
    - Stack tracking: Using a stack created from a doubly-linked list for which each page access moves the page to the top

    - Memory reference updating is expensive, requires hardware
    - No search is required

- **Second-chance/clock replacement:** Using a set of reference bits which are set when a page is referenced and reset when a page is replaced, and removing a page which was not used since the last replacement
- **Least Frequently Used (LRU) page replacement algorithm:** Removing from memory the page which has been used the least
    - Removes pages which are not used often
    - Early heavily-used pages will stay in memory
- **Most Frequently Used (MRU) page replacement algorithm:** Removing from memory the page which has been used the most
    - Pages with smaller counts are estimated to be newer and kept
    - Heavily-used pages may be removed more frequently
- **Optimal page replacement algorithm:** Removing from memory the page which will not be used for the longest period of time, upon a fault
    - Only possible in theory or when analyzing a past set of page accesses
    - Used to benchmark existing algorithms

8.3 Frame Allocation

- **Equal allocation:**
- **Proportional allocation:** Allocating frames per process relative to the size of the process
    - Formula:

$$\begin{aligned} \text{Let } m &= \text{ total number of frames} \\ \text{Let } p_i &\text{ be a given process} \\ \text{Let } s_i &= \text{ size of process } p_i \\ \text{Page allocation } a_i \text{ for } p_i &= \frac{s_i}{\sum s_i} \times m \end{aligned}$$

    - Example:

$$\begin{aligned} m &= 64 \text{ frames} \\ s_1 \text{ for process } p_1 &= 10 \\ s_2 \text{ for process } p_2 &= 127 \\ \text{Page allocation } a_1 &= \frac{s_1}{\sum s_1} \times m \\ &= \frac{10}{10+127} \times 64 \\ &\approx 5 \end{aligned}$$

- **Priority allocation:** Allocating frames per process relative to the priority of the process

- **Global replacement:**
- **Local replacement:**
- **Locality model:** Keeping in memory the pages necessary for the current segment of code
- **Working set:**
  - Managed with an interval timer and set of reference bits (per page in memory), which is set when a page is referenced
- **Thrashing:** Situation where a process is starved of necessary frame allocations and must constantly swap pages

8.4 Kernel Memory Allocation

- **Buddy system:** Allocating memory from fixed-size contiguous segments with size to the power of 2
  - May have up to 50% internal fragmentation
- **Slab allocator:** Allocating memory from precreated spaces for kernel data structures
  - **Cache:** Paged instantiation space in memory for an object
  - **Slab:** Contiguous memory which contains multiple caches
  - Memory allocation is fast and has no fragmentation
- **Memory mapping:** Mapping a file to a memory address space so modifications have the behaviour of ordinary memory accesses

9 Filesystems

- Cyclic links can cause infinite loops during searches
- Mounting:
  - –