Topic 6 - Process Synchronization

Process Synchronization

If a shared resource is being modified by more than one thread concurrently, then unexpected results may occur!

- The problem is that concurrent threads access a shared resource without any synchronization, and this creates a race condition (output depends on timing)
- We need mechanisms for controlling access to shared resources when working with concurrency
- Synchronization is necessary for any shared data structure

What resources are shared among threads?

- Local variables and **NOT** shared (data on the threads stack)
- Global variables are shared
- Dynamically allocated objects are shared
    - Stored in the heap

We want mutual exclusion when accessing shared resources

- When one thread is using a shared resource, other threads must be excluded from accessing the same resource until the first thread is done with it

The part of the code where the shared resource is accessed is called the **critical section**

- Only one thread can execute in the critical section at a time, all other threads are forced to wait on entry

Structure

```
<Entry Code>

Critical Section

<Exit Code>

Remainder Section
```

```
int deposit(account, amount){
        <Entry Code>
        balance = get_balance(account);
        balance += amount;
        put_balance(account, balance);
        <Exit Code>
        return balance;
} //end-deposit
```

Critical Section

- At most one thread is in the critical section at once
- If thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
- If thread T is waiting to enter the critical section, then T will eventually enter the critical section (no starvation)

Mechanisms for Synchronization in Concurrent Programs

Mutex Locks (spinlocks, blocking locks)

- A basic primitive for ensuring mutual exclusion in multi-threaded code

Semaphores

- A higher-level synchronization primitive used for both mutual exclusion and coordination between threads or processes

Mutex Locks

- A synchronization primitive used to protect critical sections (CS), ensuring only one thread can access a shared resource at a time with the following two operations:
  - acquire()/lock(): a thread calls this before entering the CS
  - release()/unlock(): a thread calls this after leaving the critical section
  - Threads must pair up these calls acquire() -> release() / **lock() -> unlock()**

Example:

```
int lock = 0; // unlocked
int deposit(account, amount){
      lock(lock);
      balance = get_balance(account);
      balance += amount;
      put_balance(account, balance);
      unlock(lock);
      return balance;
}
```

Spinlocks: Pros & Cons

Pros:

- Can be implemented in user-space (no need to trap to the kernel)
- Useful in multi-processor environments

Cons:

- Horribly wasteful in single CPU environment
- If a thread is spinning on a lock, it is simply wasting CPU cycles

Mutex Locks: Ideal Implementation

When acquire() is called:

- Spin for a short amount of time to get the lock
- If you get the lock during this time, enter the CS
- If you don't get the lock, trap to the kernel to release the CPU and sleep on the mutex's waiting queue

When release() is called:

- If there is a thread sleeping on the mutex's waiting queue, trap to the kernel and wake one of these threads
- Otherwise simply set the mutex to 0 (unlocked)

POSIX pthread_mutex functions work exactly this way!

Pthread Mutex Syntax

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

- **pthread_mutex_init(&mutex, NULL);**

int pthread_mutex_lock(pthread_mutex_t *mp); // function to lock

- **pthread_mutex_lock(&mutex);**

int pthread_mutex_unlock(pthread_mutex_t *mp); //function to unlock

- **pthread_mutex_unlock(&mutex);**

Recursive Locking

06-ProcessSynchronization/ex6.c

```
pthread_mutex_t mutex;

void functionB() {
    pthread_mutex_lock(&mutex);
    printf("Inside functionB\n");
    pthread_mutex_unlock(&mutex);
}

void functionA() {
    pthread_mutex_lock(&mutex);
    printf("Inside functionA\n");
    functionB();  // Calls another
    pthread_mutex_unlock(&mutex);
}
```

Semaphores

- A synchronization primitive that is used for both mutual exclusion and coordination between threads or processes
- An integer variable that is manipulated atomically through two operations: wait(down, P) and post(up, V)
    - wait(semaphore): decrements the semaphore value
        - Blocks the thread if semaphore value is non-positive until becomes open
    - post(semaphore): increments the semaphore value
        - Unblocks a waiting thread (if any)

There are two types of semaphores

- Binary Semaphore (a.k.a mutex semaphore)
    - Takes the values of 0 or 1
    - Use as a blocking mutex lock: Initialize to 1 to allow mutual exclusion
    - Use for thread/process coordination: Initialize to 0 to act as a signaling mechanism. This use is NOT possible with mutex locks
- Counting Semaphore
    - Takes arbitrary values
    - Represents a resource with many units available
    - Counter is initialize to N, where N is the number of available units
    - Allows thread/processes to proceed as long as more units are available
    - Each wait() decrements the counter, and each post() increments it

Initialized as: sem_t sem;
Functions:
int sem_init(sem_t *sem, int pshared, unsinged int value); //assign initial value

- Initialize 0: sem_init(&sem, 0, 0);
- Initialize 1: sem_init(&sem, 0, 1);
- Initialize N: sem_init(&sem, 0 N); where N >= 0

int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_destroy(sem_t *sem);

Use Case 1: Mutual Exclusion

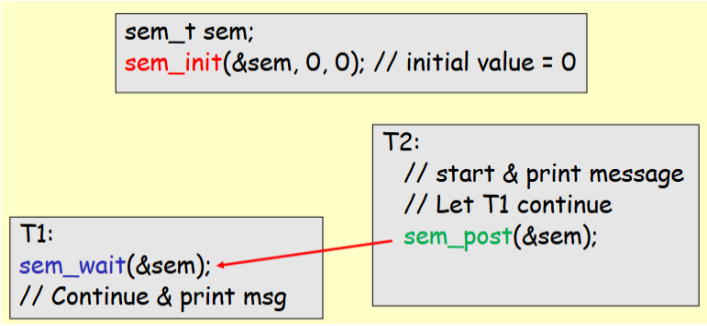
- Declare a binary semaphore & initialize it to 1
- Call sem_wait() before entering the critical section
- Call sem_post() when leaving the critical section

```
sem_t sem;
sem_init(&sem, 0, 1);
sem_wait(&sem);
//Critical section code
sem_post(&sem);
```

Use Case 2: Provide Coordination

Assume you have two threads T1 & T2

- We want T2 to run first & print a message
- Then we want T1 to continue execution & print a message
- To do this: declare a **binary semaphore** & initialize it to 0
- T1 calls sem_wait() to stop at the rendezvous point
- T2 calls sem_post() to let T1 continue



Use Case 3: Allow a certain number of threads inside the critical section

Assume you have an application where you have 3 resources of a certain kind

- You want to have a critical section where there could be up to three threads inside the CS at the same time
- 4th thread trying to enter the critical section must block at the entry until one of the threads inside the CS leaves
- To do this: create a counting semaphore with an initial value of 3
- Threads simply do sem_wait() at Entry, sem_post() at Exit
- This is the molecule example!

Deadlocks & Starvation

Semaphores can lead to deadlock if not used properly

Deadlocks - two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Starvation - indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended

Semaphores vs Mutex Locks

Aspect	Mutex Locks	Semaphores
Purpose	Mutual exclusion mechanism	Signaling mechanism
Counter Value	Always binary (unlocked or locked)	Can be 0, 1 (binary semaphore), or greater than 1 (counting semaphore)
Ownership	Owned by the thread that locks it	Not owned by a specific thread
Blocking	Lock operation blocks if already locked	Wait operation blocks if value = 0
Complexity	Simpler: single thread access only	Can manage multiple threads simultaneously with counting semaphores
Use Cases	Protecting critical sections	Synchronization and coordination
Deadlock Potential	Can occur if unlock is not managed properly	Can occur if used improperly

Monitors

Mutexes & semaphores are great to implement critical sections and coordinate threads, but they are hard to use & easy to make mistakes

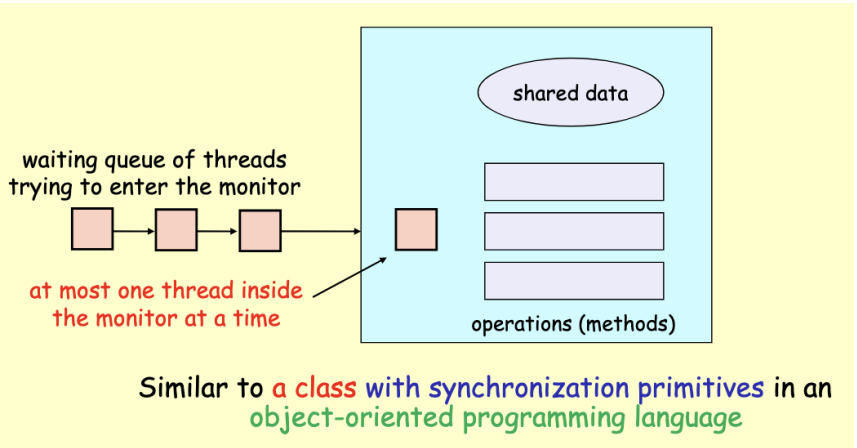
- Can easily write code that leads to deadlocks or not work at all
- Hard to debug
- Monitors with condition variables take away some of the programmer's responsibility

A programming language construct that supports controlled access to shared data

- Synchronization code added by compiler, enforced at runtime
- Programmer is freed from using semaphores in the correct manner
- If a second thread tries to enter a monitor method, it blocks until the first has left the monitor

A software construct (a class) that encapsulates:

- Shared data
- Methods that operate on the shared data
- Synchronization and coordination between concurrent threads that invoke those methods



Implementation of Monitor Methods

A mutex lock is associated with each monitor instance (object)

- The body of each method is converted into a critical section with this mutex
- That is, each method P is implemented as follows:

```
mutex_t mutex; // Monitor lock

P(){
      lock(mutex);

      <Body of P>

      unlock(mutex);
}
```

Java Synchronized Methods

- Protected by a “reentrant” mutex lock
  - This means that if a synchronized method calls another synchronized method, deadlock will NOT occur
  - When you mark a method synchronized, the entire body becomes a CS

```
class Account {
      private int balance;
      public Account (int initialDeposit) {
            balance = initialDeposit;
      }
      public synchronized int getBalance() {
            return balance;
      }
      public synchronized void deposit(int amount){
            balance += amount;
      }
      public synchronized void withdraw(int amount) {
            if (amount > balance) return;
            balance -= amount;
      }
}
```

```
void method1() {
   synchronized (this) {
      statement 1;
      statement 2;
      statement 3;
      statement 4;
      statement 5;
      statement 6;
   }
}
```

```
void method1() {
   statement 1;
   statement 2;
  synchronized (this) {
      statement 3;
      statement 4;
      statement 5;
   }
   statement 6;
}
```

Monitor and Condition Variables

Once inside a monitor, a thread may discover it can't continue

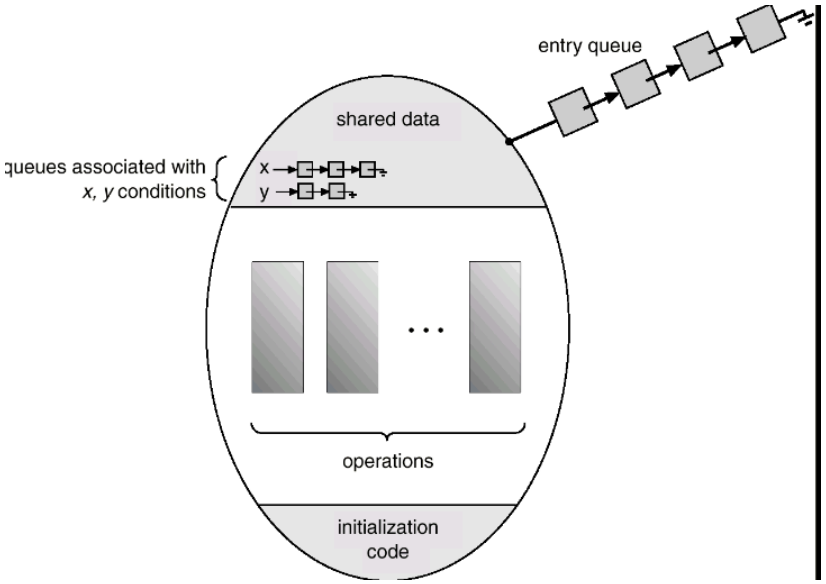
- A producer finds Q is full, or a consumer finds Q is empty and may wish to release the monitor lock and wait for the condition to become true
- Release the lock allows some other waiting thread to enter the monitor

Threads need a place to sleep until the condition becomes true

- Condition variables provided within the monitor server this purpose
- Thread waits on a condition variable, releases the monitor lock so that other threads can continue and enter the monitor
- When the condition becomes true, another thread executes "signal" on the condition variables to wake up the waiting thread

By default there is one condition variable associated with the monitor (object) lock

- When you do wait(), you go to sleep until somebody executes a notify() or notifyAll()



Main Operations

wait(c)

- Release the monitor lock, so another thread can get in
- Sleep on the waiting queue of the condition variable
- Wait for another thread to signal the condition and wake the thread up

signal(c)

- Wake up at most one waiting thread & move it to the ready queue
- If no waiting thread, signal is lost (no thread to wake up)
- This is different that semaphores: no history

on

```
synchronized void method1() {
    while (!condition)
        wait();
}
```

```
synchronized void method2() {
    // change value in condition test
    notify();
}
```

Multiple Condition Variables

You can use multiple condition variables instead of the single one provided by the monitor using:

- Lock lock = new ReentrantLock();
- Condition condVar1 = lock.newCondition();
- Must create your own lock for the class and not use the synchronized keyword if you want multiple condition variables

Example

```
public class Foo3 {
    private boolean condition1 = false;
    private boolean condition2 = false;
    private final Lock lock = new ReentrantLock();
    private final Condition condVar1 = lock.newCondition();
    private final Condition condVar2 = lock.newCondition();
```

```
public void threadT1() throws
    lock.lock();
    try {
        System.err.println("T1
        while (!condition1) {
            condVar1.await();
        }
        System.out.println(x:"
    } finally {
        lock.unlock();
    }
}
```

```
public void threadT2() throws
    lock.lock();
    try {
        System.err.println("T2
        while (!condition2) {
            condVar2.await();
        }
        System.out.println(x:"
    } finally {
        lock.unlock();
```

```
public void threadC1() throw
    lock.lock();
    try {
        condition1 = true;
        System.out.println(x
        condVar1.signal();
    } finally {
        lock.unlock();
    }
}

public void threadC2() throw
    lock.lock();
    try {
        condition2 = true;
        System.out.println(x
        condVar2.signal();
    } finally {
        lock.unlock();
    }
}
```

Practice Problems: Process Synchronization

1. What's a Critical Section? What is mutual exclusion? Write a simple code to show how you would achieve mutual exclusion in a critical section using a mutex lock.

The critical section is a part of the code where at most on thread may be executing at any given time. Mutual exclusion is when one thread is using a shared resource, all other threads must be **excluded** from using that same resource.

2. What is atomic instruction? In Intel architecture, what's the name of the atomic instruction used to implement a mutex lock?

An atomic instruction is one that **cannot** get interrupted. In intel architecture, the name of the atomic instruction used to implement a mutex lock is Compare & Change.

3. What's Test-and-Set instruction? What is it used for? Briefly describe.

Tests the value of a memory location and sets it to a new value in atomically. First loads 0 unlocked, then attempts to lock by setting to 1, but only if the tests passes and its 0, otherwise the lock was already set. It is used to implement locks and achieve mutual exclusion.

4. Briefly explain why spinlocks are not appropriate for single processor systems yet are often used in multiprocessor systems.

Constantly wasting CPU cycles so that a thread can check if another thread is done with a lock. With multiprocessor systems, you can make this checking happen on one core alone while have execution continue on another core.

5. What's race condition? Briefly explain.

A race condition is when multiple threads try to access the same resource at the same time. This causes the output to be nondeterministic and uncertain, as the results depends on timing of threads.

6. List 4 requirements of a critical section.

1. Mutual Exclusion
2. Progress (no dead locks)
3. No starvation (Bounded Waiting)
4. Performance (small overhead)

7. List 2 reasons why it is a bad idea to implement locks by disabling interrupts?

One reason is this will affect important system activities. Another reason is that in multi-core processors, threads running on another core would still be able to run, potentially breaking mutual exclusion.

8. What's a deadlock? How does it occur? Briefly explain.

If a thread fails to unlock a thread it creates a deadlock so all other processes are stuck in a waiting cycle and are starved since that process will never release

9. What's a condition variable. Why and where do you need them. Briefly explain.

A condition variable is a tool that allows threads to sleep until a certain condition is true. This is used in multi threaded applications to synchronize threads together, and this is important because sometimes a thread cannot continue until something else occurs.

10. Compare and contrast a mutex lock to a binary semaphore in terms of purpose and ownership.

With mutex's the thread has ownership over the lock, and with binary semaphores the thread does not have ownership over it. Both accomplish mutual exclusion, but mutex locks are safer because since there is no ownership in semaphores, another thread not in the CS can unlock the semaphore.

11. Is it possible for one thread to lock a mutex lock and another thread to unlock it? Why or why not. Briefly explain.

No

12. Is recursive locking possible with mutex locks? Is it possible with binary semaphores? Why or why not. Briefly explain.

Yes this is possible with mutex locks as long as the same thread that does the locking does the unlocking. This is not possible with binary semaphores because they do not have the same ownership properties.

13. What's the purpose of recursive locking and how do you do it? Briefly explain.

Recursive locking allows a thread that currently holds the lock to acquire it again without being blocked. Without recursive locking the thread would deadlock itself, which is why we have recursive locking! You do this by using a special type of lock called a recursive mutex lock, and the lock accomplishes this by keeping a count of how many times the lock was used and only releasing once the count goes back to 0 (unlock).

14. Briefly explain what priority inversion is and how unbounded priority inversion occurs.

15. Briefly explain how priority ceiling protocol solves the priority inversion problem.

16. Briefly explain how priority inheritance protocol solves the priority inversion problem.

23. Write a monitor that implements an alarm clock that enables a calling program to delay itself for a specified number of time units (ticks). You may assume the existence of a real hardware clock, which invokes a procedure "Tick" in your monitor at regular intervals.

```
public class AlarmClock{
   // <declarations, initialization, etc.>
   int ticks = 0;
   public synchronized void Wait(int numOfTicksToWait) throws InterruptedException {
       //<** YOUR CODE HERE **>
       int wakeupTicks = ticks + numOfTicksToWait;
       while(ticks < wakeupTicks){
           wait();
       }
   } //end-Wait
   public synchronized void Tick(){ /* Tick is automatically called by the hardware */
   // <** YOUR CODE HERE **>
       ticks += 1;
       notifyAll();
   } //end-Tick
} //end-Monitor
```

31. Fill in the details of the following Monitor that implements a thread-safe Queue using a monitor with the following constraints: When you want to “add” an item and the queue is full, you must wait until an empty slot becomes available. When you want to “remove” item and the queue is empty, you must wait until a slot becomes full.

This solution uses only one condition variable / lock

```
public class Queue {
   int front = 0; // Initializations
   int rear = 0;
   int numItems = 0;
   int n = 10;
   int q[];
   // At most N items
   // Other initializations below (if necessary)
   // Insert an element to the rear of the Queue
   public synchronized void add(int item) throws InterruptedException {
       while(numItems  == n){
           wait();
       }
       q[rear] = item;
       rear += 1;
       rear %= n;
       numItems += 1;
       notifyAll();
   } //end-add

   // Remove an element from the front of the Queue
   public synchronized int remove() throws InterruptedException {
       while(numItems == 0){
           wait();
       }
       int val = q[front];
       front += 1;
       front %= n;
       numItems -= 1;
       notifyAll();
       return val;
   } //end-remove
} //end-Monitor
```

```
public class Queue {
   int front = 0; // Initializations
   int rear = 0;
   int numItems = 0;
   int n = 10;
   int q[];
   // At most N items
   // Other initializations below (if necessary)
   Lock lock = new Reentrantlock();
   Condition canAdd = lock.newCondition();
   Condition canSub = lock.newCondition();

   // Insert an element to the rear of the Queue
   public void add(int item) throws InterruptedException {
       lock.lock();
       while(numItems  == n){
           canAdd.await();
       }
       q[rear] = item;
       rear += 1;
       rear %= n;
       numItems += 1;
       canSub.signal();
       lock.unlock();
   } //end-add

   // Remove an element from the front of the Queue
   public int remove() throws InterruptedException {
       lock.lock();
       while(numItems == 0){
           canSub.await();
       }
       int val = q[front];
       front += 1;
       front %= n;
       numItems -= 1;
       canAdd.signal();
       lock.unlock();
       return val;
   } //end-remove
} //end-Monitor
```

18. Consider a global variable "g" with initial value 0, that is manipulated by multiple threads. There are two types of threads: Adders and Subtractors. The threads obey the following rules:

1. An Adder arrives and sees the value of "g" to be less than 5, it simply increments the value of "g" by one and leaves.
2. An Adder arrives and sees the value of "g" to be equal to 5, it waits until the value of "g" drops below 5.
3. A Subtractor arrives and sees the value of "g" to be greater than 0, it simply decrements the value of "g" by one and leaves.
4. A Subtractor arrives and sees the value of "g" to be equal to 0, it waits until the value of "g" goes above 0. You are asked to implement this synchronization problem using a monitor. Your monitor must have two methods, Add and Subtract, called by Adders and Subtractors respectively.

Implement the above monitor in Java.
One cond

```java
public class GlobalVar {
   int g = 0;

   public synchronized void Adder() throws InterruptedException {
       while(g == 5){
           wait();
       }
       g += 1;
       notifyAll();
   }

   public synchronized void Subtractor() throws InterruptedException {
       while(g == 0){
           wait();
       }
       g -= 1;
       notifyAll();
   }
}
```

```
class GlobalVar{
	int g = 0;

	mutex_t mutex = PTHREAD_MUTEXT_INITIALIZER;
	cond_t canAdd;
	cond_t canSub;

	public void Adder(){
		lock(&mutex);
		while(g == 5){
			cond_wait(&canAdd, &mutex);
		}
		g += 1;
		cond_signal(&canSub);
		unlock(&mutex);
	}

	public void Subtractor(){
		lock(&mutex);
		while(g == 0){
			cond_wait(&canSub, &mutex);
		}
		g -= 1;
		cond_signal(&canAdd);
		unlock(&mutex);
	}
```

Multiple Conds

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class GlobalVarMult {
   int g = 0;
   Lock lock = new ReentrantLock();
   Condition canAdd = lock.newCondition();
   Condition canSub = lock.newCondition();

   public void Adder() throws InterruptedException {
       lock.lock();
       while(g == 5){
           canAdd.await();
       }
       g += 1;
       canSub.signal();
       lock.unlock();
   }
   public void Subtractor() throws InterruptedException {
       lock.lock();
       while(g == 0){
           canSub.await();
       }
       g -= 1;
       canAdd.signal();
       lock.unlock();
   }
}
```

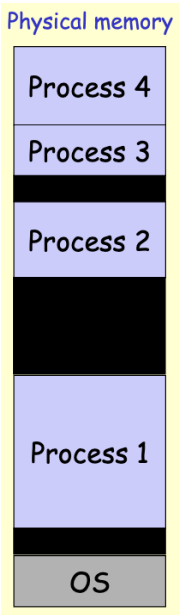
Topic 7 - Memory Management

Purpose?

- A process needs to be loaded into memory before it can run!
- So, the OS must manage the available memory
    - Allocate enough memory to hold the process's code and data when the process is created
    - Reclaim a process's memory when the process terminates
    - Protect one process's memory space from modification by other processes in the system (memory protection)
    - Keep track of which parts of memory are in use

History

- Single-Tasking Systems
    - Run a single program at a time (MS-DOS model)
        - OS loads the program, runs it, unloads it
    - An OS with one user process
- Multiprogramming (Multi-tasking)
    - Keep multiple processes in memory at once to increase utilization
        - To overlap I/O and computation
    - Variable partitioning



Memory Allocation Algorithms in Variable Partitioning

- First-fit: Allocate the **first hole** that is big enough
    - Example: Incoming Processes P4(4 KB), P5(1 KB)

    P1(3 KB) | Hole(5 KB) | P2(7 KB) | Hole(10 KB) | P3(3 KB) | Hole(4 KB)
    P4(4 KB) comes in
    P1(3 KB) | P4(4 KB) | Hole(1 KB) | P2(7 KB) | Hole(10 KB) | P3(3 KB) | Hole(4 KB)
    P5(1 KB) comes in
    P1(3 KB) | P4(4 KB) | P5(1 KB) | P2(7 KB) | Hole(10 KB) | P3(3 KB) | Hole(4 KB)

- Best-fit: Allocate the **smallest hole** that is big enough; must search the entire list, unless ordered by size. Produces the smallest leftover hole

    P1(3 KB) | Hole(5 KB) | P2(7 KB) | Hole(10 KB) | P3(3 KB) | Hole(4 KB)
    P4(4 KB)
    P1(3 KB) | Hole(5 KB) | P2(7 KB) | Hole(10 KB) | P3(3 KB) | P4(4 KB)
    P5(1 KB)
    P1(3 KB) | P5(1 KB) | Hole(4 KB) | P2(7 KB) | Hole(10 KB) | P3(3 KB) | P4(4 KB)

- Worst-fit: Allocate the **largest hole**; must also search the entire list. Produces the largest leftover hole
  P1(3 KB) | Hole(5 KB) | P2(7 KB) | Hole(10 KB) | P3(3 KB) | Hole(4 KB)
  P4(4 KB)
  P1(3 KB) | Hole(5 KB) | P2(7 KB) | P4(4 KB) | Hole(6 KB) | P3(3 KB) | Hole(4 KB)
  P5(1 KB)
  P1(3 KB) | Hole(5 KB) | P2(7 KB) | P4(4 KB) | P5(1 KB) | Hole(5 KB) | P3(3 KB) | Hole(4 KB)

Variable Partitioning with Physical Addresses ~1960

CPUs used physical addresses to access memory

- The compiler assumed that the executable would be loaded at a fixed address, usually 0!
- The compiler generated static memory addresses for global variables and function calls based on this assumption
- If the process wasn't loaded at the expected address, these memory references would be incorrect

Solution:

- Compilers generated relocation records in the executable, which the OS used during loading
- The OS loader adjusted memory references using the relocation records to ensure correctness

Problems:

- Complex: Relocation records make loading processes slow and error-prone
- Unsafe: Processes could overwrite each other's memory, leading to crashes and security issues

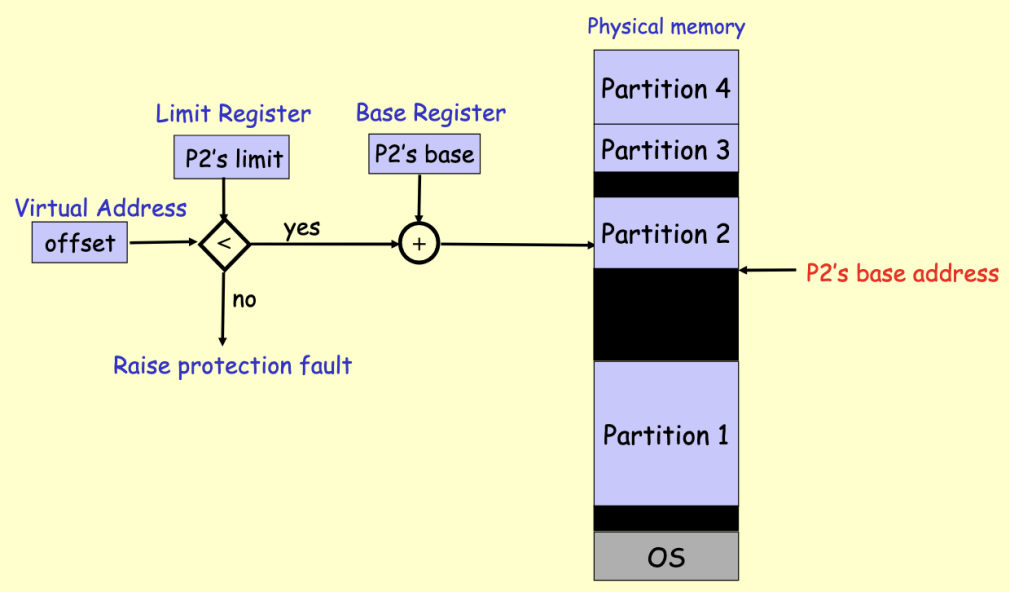
MMUs & Logical (Virtual) Addresses

To solve the problems with physical addresses, CPUs with simple Memory Management Units (MMUs) and logical (virtual) addresses were introduced

- CPU & processes started using logical (virtual) addresses
- Logical addresses are independent of the location of the process in RAM
- MMU must intercept every memory address generated by the CPU and convert the logical address into a physical address

Simple MMU: Base & Limit Registers:

- The **base** registers holds the starting physical address of a process's memory space
- The **limit** register defines the size of the process's memory space



Benefits?

Flexibility:

- Processes could be loaded into any available memory location without modifying their code (no need for relocation records & address adjustment)
- The base register would be set up accordingly
- You can move the process to a different location without any problems

Memory Protection:

- The limit register prevents processes from accessing memory outside of their allocated space providing memory protection among different processes

Process Swapping (1971 ~ 1983)

Consider a case where a higher priority process arrives, but there is not enough space to load it

- The OS can simply have the higher process wait in the queue of processes waiting to be admitted to the system until enough space becomes available
- Swap an entire process temporarily out of memory to a **backing store** and then bring it back later into memory to continue execution
- Backing store: fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Major part of swap time is the transfer time; which is directly proportional to the amount of memory swapped

Variable Partitioning with MMU and Logical Addresses

External Fragmentation is a big problem

- The entire process must still fit into memory and be stored consecutively

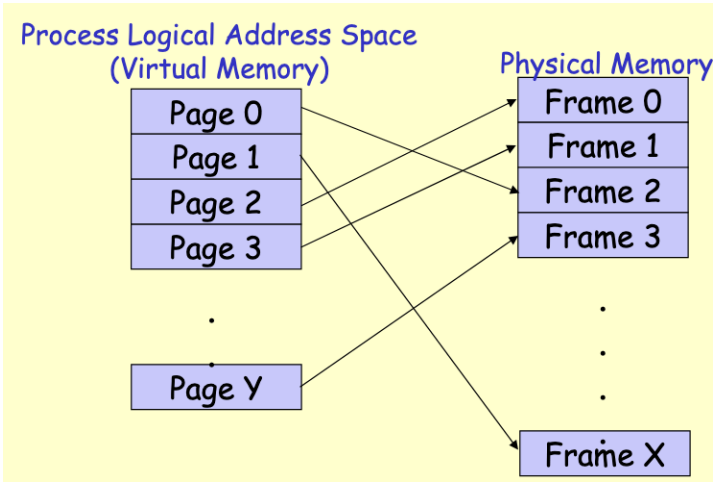
Segmentation was introduced

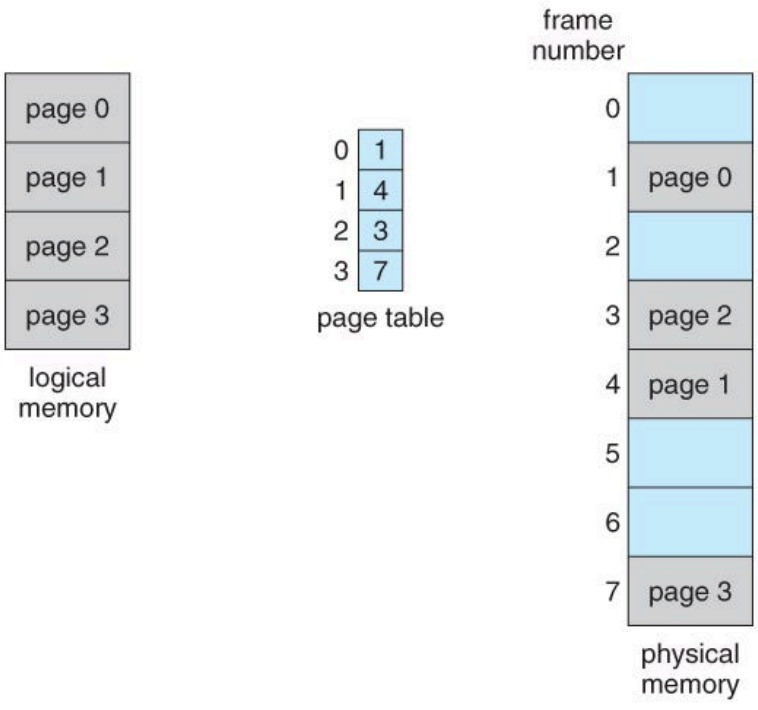
- Divide a process into multiple segments: **Code, Data, Stack**
- Can store each segment into a different memory location
- Instead of one base / limit pair, each segment has its own base & limit registers
- Segmentation was still NOT the ultimate solution

Paging (Modern Technique)

Solves the external fragmentation problem by using fixed sized units (page/frame) in both virtual and physical memory

- Page size is a power of 2, e.g., 512, 1024, 8192 etc.
- OS keeps track of all free frames
- To run a program of size n pages, **need to find n free frames** and load the program

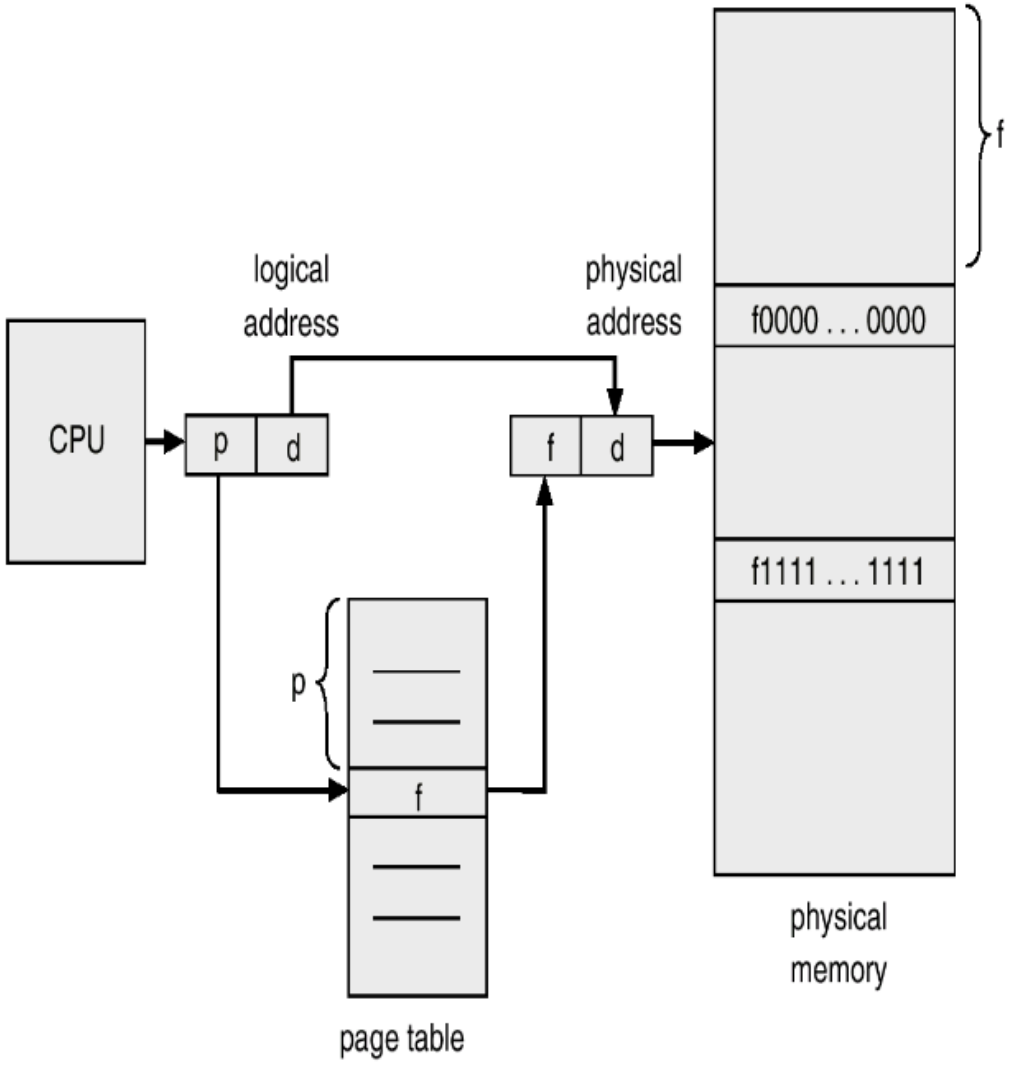




Address Translation with Paging

Translating virtual addresses:

- A virtual address has two parts: page number (PN) and offset
- PN is an index into a page table (PT)
- Each PT entry (PTE) contains frame number (FN)
- Physical address is FN::offset
- Page tables are managed by to OS
- Each process has its own page table



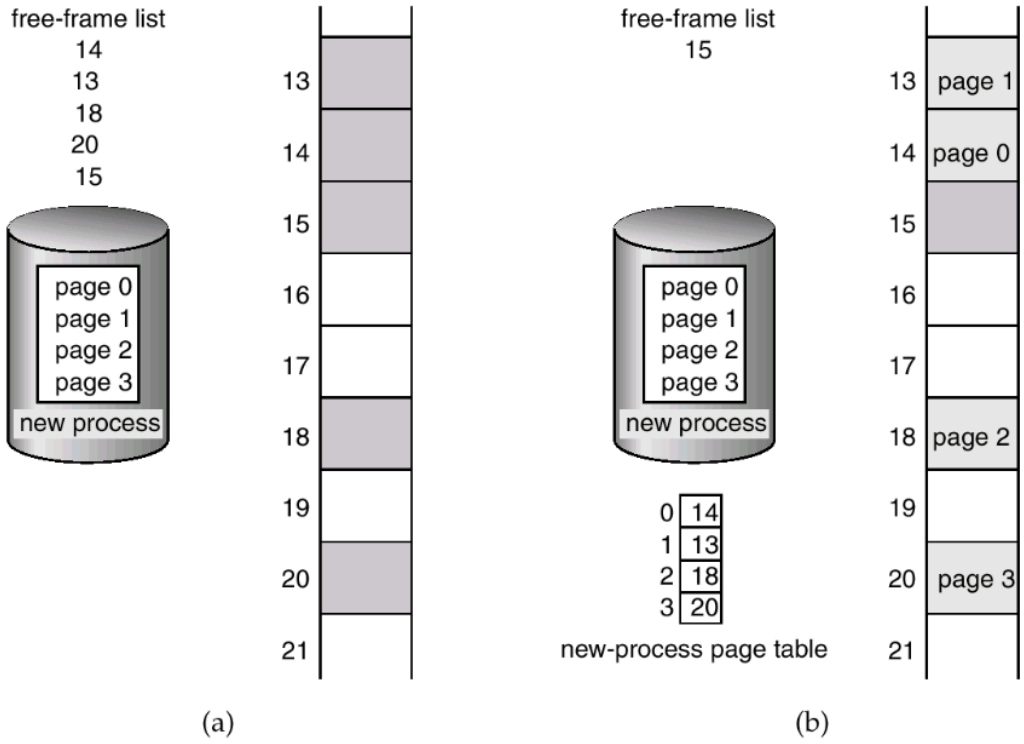
Paging Example

Assume 32-bit addresses

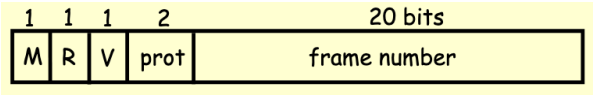
- Assume page size is 4 KB (4096 bytes, or 2^12 bytes
- PN is 20 bits long (2^20 PNs), offset is 12 bits long

Let's translate virtual address 0x13325328

- PN (page number) is 0x13325, and offset is 0x328
- Assume page table entry 0x13325 contains value 0x03004
    - Frame number is 0x03004
    - PN 0x13325 maps to FN 0x03004
- Physical address = FN::offset = 0x03004328



Page Table Entries (PTEs)



- The **modify bit** says whether or not the page is dirty
    - It is set when a write to the page has occurred
- The **reference bit** says whether the page has been accessed
    - It is set when a page has been read or written to
- The **valid bit** says whether or not the PTE can be used
    - Says if a virtual address is valid & the corresponding page is present in physical memory
    - It is checked each time a virtual address is used. If invalid, a page fault is generated
- The protection bits control which operations are allowed
    - Read, write, execute

- The frame number determines the physical page
    - Physical page state address = FN << (#bits/page)

Advantages

Easy to allocate physical memory

- Physical memory is allocated from free list of frames
    - To allocate a frame, just remove it from its free list
- External fragmentation is NOT a problem!
- Easy to “page out” frames of the process
    - All frames are the same size (page size)
    - Use valid bit to detect references to “paged-out” pages
    - Also, page sizes are usually chosen to be convenient multiples of disk block sizes

Disadvantages

Can still have internal fragmentation

- Process may not use memory in exact multiples of pages

Memory reference overhead

- 2 references per address lookup (page table, then memory)
- Solution: use a hardware cache to absorb page table lookups
    - Translation lookaside buffer (TLB)

Memory required to hold page tables can be large

- Need one PTE per page in virtual address space
- 32 bit address space with 4 KB pages = 2^20 PTEs = 1,048,576 PTEs
- 4 bytes/PTE = 4 MB per page table
- Solution: page the page tables!!

Making Page Tables Manageable

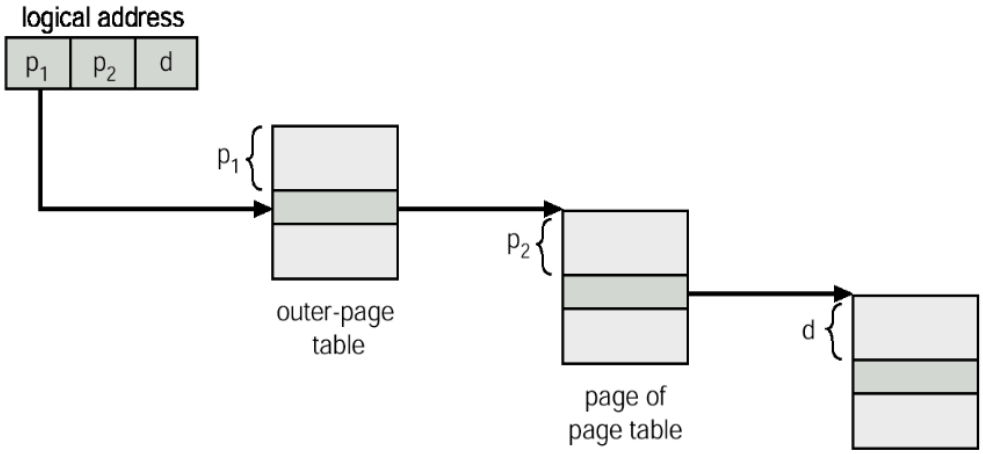
Size of a page table for 32-bit address space with 4KB pages was 4MB (too much overhead)
Reduce?

- Observation: only need to map the portion of the address space that is actually being used (tiny fraction of address space)
    - Only need page table entries for those portions
- How can we do this?
    - Make the page table structure dynamically extensible
        - With level of indirection: two-level page tables

Two-Level Page Tables

With two-level PTs, virtual addresses have 3 parts:

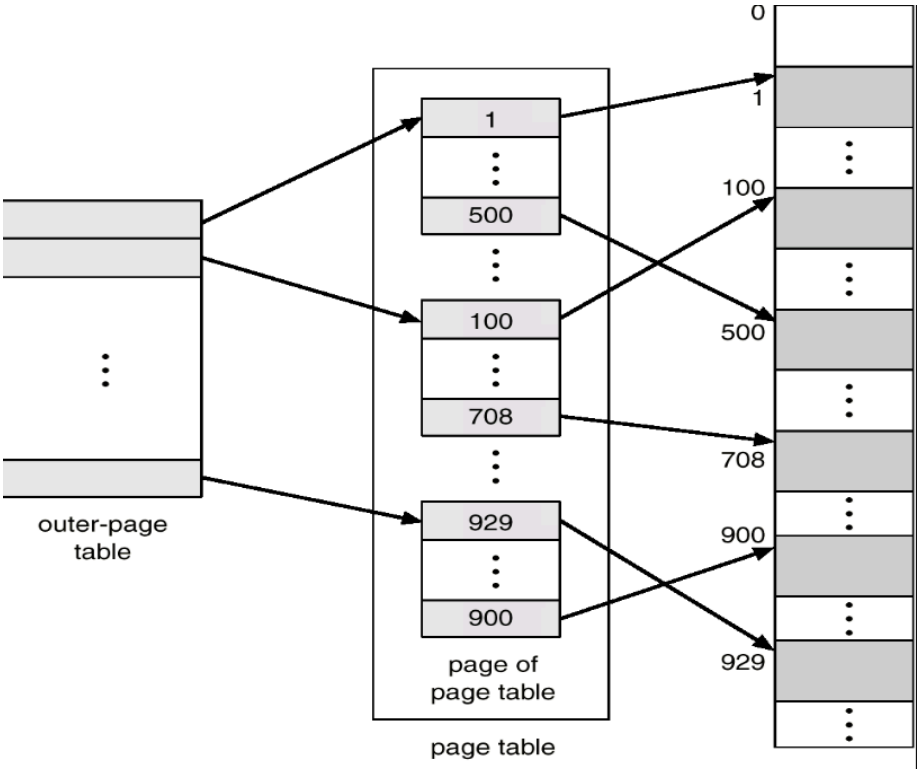
- Master page number, secondary page number, offset
- Master PT maps master PN to secondary PT
- Secondary PT maps secondary PN to page frame number
- offset + FN = physical address



Example:
32-bit machine, 4KB pages, 4 bytes/PTE

- How many bits in offset? Need 12 bits for 4 KB
- Inner page table must fit into a single frame: 4KB/4 bytes = 1024 PTE
    - So, 10-bits for the inner page table
- That leaves use with 10-bits for the outer (master) page table, which also fits into a single frame
- Thus a logical address is as follows -> Where p1 is an index into the outer page table, and p2 is an index into the inner page table

page number		page offset
p_1	p_2	d
10	10	12



Making it efficient

One-level page table scheme doubled the cost of memory lookups

- One lookup into page table, a second to fetch the data

Two-level page tables triple the cost!

- Two lookups into page table, a third to fetch the data

How can we improve?

- Goal: make fetching from a virtual address about as efficient as fetching from a physical address
- Solution: use a hardware cache inside the CPU
    - Cache the virtual-to-physical translation in the hardware
    - Called a Translation Lookaside Buffer (TLB)
    - TLB is managed by the memory management unit (MMU)

Translation Lookaside Buffer (TLB)

- Is a fully associative cache (all entries searched in parallel)
- Translates virtual page number → Page Table Entry (PTE)
- Can be done in single machine cycle
- With PTE + offset, MMU can directly calculate the Physical Address

Misses

- When a TLB miss occurs, the MMU walks the page tables to load the required Page Table Entry (PTE) into the TLB
    - If the TLB is full, a victim PTE is evicted based on the TLB replacement policy, which is implemented in hardware

TLB & Context Switches

- During a process context switch, the OS switches from one process to another
    - To avoid using stale translation, the OS flushes the TLB, invalidating all entries
    - This forces the new process to repopulate the TLB from scratch, causing TLB misses and slowing down memory access

Effective Memory Access Time

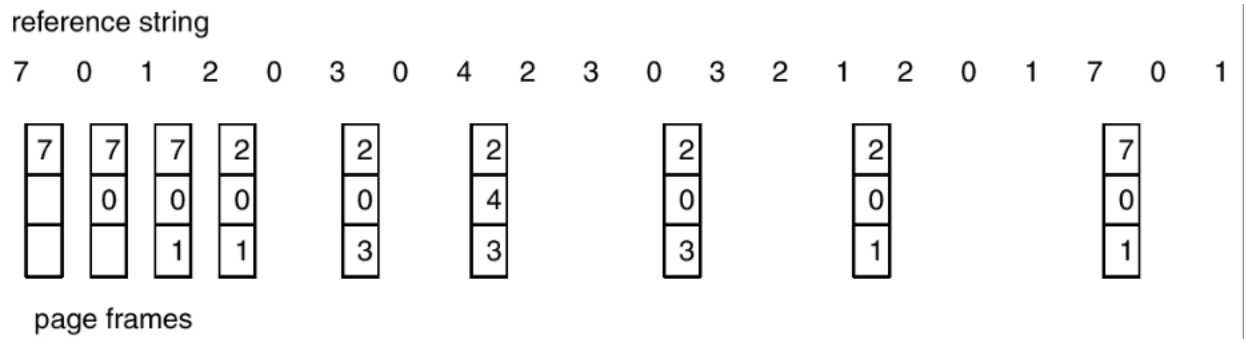
- Effective access time for n level page tables
- Associative Lookup = ξ time unit
- Assume memory cycle time is β time unit
- Hit ratio - percentage of times that a page number is found in the associative registers
    - Ratio related to number of associative registers
- Hit ratio = α
- Effective Access Time (EAT)
    - $EAT = (\beta + \xi)\alpha + ((n+1)\beta + \xi)(1 - \alpha)$

Page Replacement Algorithms

Belady's (Optimal) Algorithm

Provably optimal lowest fault rate

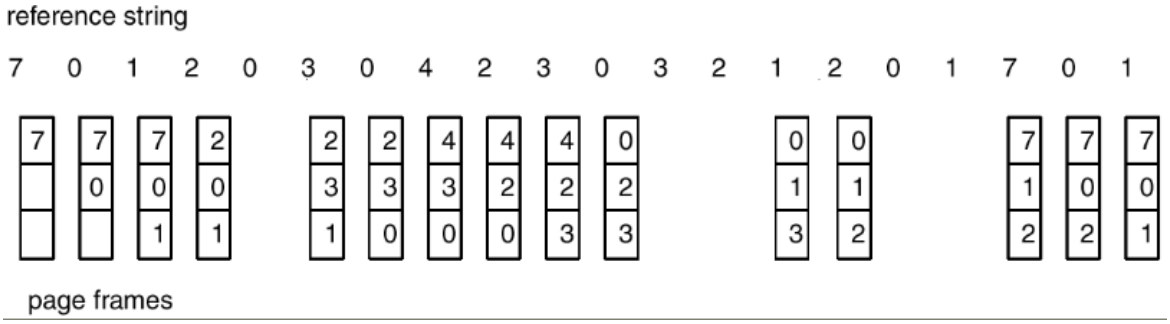
- Pick the page that won't be used for longest time in future
- Used as a measurement to compare other algorithms
    - If Belady's isn't much better than yours, yours is pretty good



FIFO

FIFO (First in First out) is obvious, and simple to implement

- When you page in something, put in at the tail of the queue
- On eviction, throw away the page at the head of the list



FIFO suffers from Belady's Anomaly

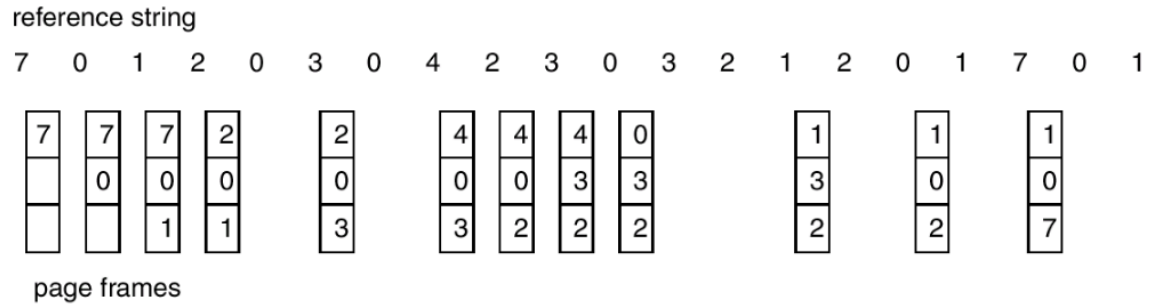
- Fault rate might increase when algorithm is given more physical memory

Least Recently Used (LRU)

LRU uses reference information to make a more informed replacement decision

- Idea: past experience gives us a guess of future behavior
- On replacement, evict the page that hasn't been used for the longest amount of time

LRU looks at the past, Belady's algorithm looks at the future



1. What's variable partitioning in memory management? Briefly explain.

Variable Partitioning in memory management is when a process arrives you reserve a partition exactly as large as the process's memory needs. Because the holes can be of any size, you can still suffer external fragmentation.

2. In early architectures, CPUs used to generate physical addresses to access RAM. What kind of problems did physical addresses create and how did memory management mechanisms solve these problems?

Using physical addresses caused a few problems, one of them being that if the process wasn't loaded at the expected address, the memory references would be incorrect. To fix this, compilers generated relocation records. With this, the OS loader adjusted memory references using the relocation records to ensure correctness.

3. Assume you have a simple program that declares a global variable and prints its address on the screen. When you run two copies of this program simultaneously, you see that both programs print the same address on the screen. How is this possible? Briefly explain.

This is possible because what is printed is the virtual address, and the virtual address points to the same location in memory, that being the location of the global variable.

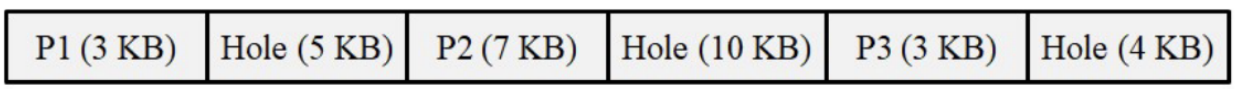
4. What's external fragmentation in memory management? Is external fragmentation a problem in paging systems? Briefly explain.

External fragmentation occurs when you use variable partitioning to load processes into memory. It is when the free slots within the memory are not big enough to hold the incoming process consecutively. Even if the TOTAL free memory is big enough to fit a new process, there may be no single contiguous slot large enough!

5. What's internal fragmentation? In what memory management method does internal fragmentation occur? Briefly explain.

Internal fragmentation is when a process does not use all of the memory allocated to it. This is because when the OS allocated memory in fixed-size chunks, a process may not need the full chunk that has been provided.

6. Consider a system that uses variable partitioning for memory management. Assume that the system has 32 KB total memory and the current snapshot of the memory is as follows:



Assume that the following 2 processes arrive in the given order: P4(4KB), P5(1KB). For each of the following memory management techniques, show the final state of the memory after the processes are allocated memory. (1) First fit, (2) Best Fit, (3) Worst fit.

First Fit

P1 (3 KB) | Hole (5 KB) | P2 (7 KB) | Hole (10 KB) | P3 (3 KB) | Hole (4 KB)
P1 (3 KB) | P4 (4 KB) | Hole (1 KB) | P2 (7 KB) | Hole (10 KB) | P3 (3 KB) | Hole (4 KB)
P1 (3 KB) | P4 (4 KB) | P5 (1 KB) | P2 (7 KB) | Hole (10 KB) | P3 (3 KB) | Hole (4 KB)

Best Fit

P1 (3 KB) | Hole (5 KB) | P2 (7 KB) | Hole (10 KB) | P3 (3 KB) | Hole (4 KB)
P1 (3 KB) | Hole (5 KB) | P2 (7 KB) | Hole (10 KB) | P3 (3 KB) | P4 (4 KB)
P1 (3 KB) | P5 (1 KB) | Hole (4 KB) | P2 (7 KB) | Hole (10 KB) | P3 (3 KB) | P4 (4 KB)

Worst Fit

P1 (3 KB) | Hole (5 KB) | P2 (7 KB) | Hole (10 KB) | P3 (3 KB) | Hole (4 KB)
P1 (3 KB) | Hole (5 KB) | P2 (7 KB) | P4 (4 KB) | Hole (6 KB) | P3 (3 KB) | Hole (4 KB)
P1 (3 KB) | Hole (5 KB) | P2 (7 KB) | P4 (4 KB) | P5 (1 KB) | Hole (5 KB) | P3 (3 KB) | Hole (4 KB)

7. Given free memory partitions of 100K, 500K, 200K, 300K and 600K (in order) how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K and 426K (in order)? Which algorithm makes the most efficient use of memory?

8. Briefly explain how memory management with variable partitioning and a simple MMU with base and limit registers works. Does this memory management scheme use physical or logical addresses?

This memory management scheme uses physical addresses! How it works is that a logical address (offset) is fed into a memory management unit (MMU), and the MMU checks if the logical address is valid for the process. The MMU contains 2 registers, base and limit. The base is the base address for the current process, and the limit contains the size of the partition. If the value in the base + offset is out of bounds of the process's limit, then a page fault is made. If it is within bounds, then the MMU directs to the process's base address which is in the base register, and adds the offset to grab the requested memory.

9. What's a virtual address? How does it differ from a physical address? Briefly explain.
A virtual address is the address used by a program to access memory. It's generated by the CPU during program execution. It does not refer to the physical location in memory, instead it is translated into physical memory by the MMU, which is an actual location in memory

10. Briefly explain what process swapping is and what it was used for? In modern paging systems, is process swapping employed? Why or why not?
Process swapping occurs when a high priority process needs to be loaded into memory to be executed but there is not enough space. In this case an entire lower priority process is removed from memory and put into a backing store, and the new process is put into memory to be executed. In modern paging systems, process swapping is NOT employed. This is because paging moves smaller memory units (pages) and **not** the entire process.

11. Briefly explain the motivation behind segmentation for memory management and how it works?
The motivation behind segmentation in memory management was to avoid external fragmentation. Before segmentation, the entire process need to be loaded into main memory in a contiguous memory space, which made it so that large processes could not be loaded when there was not enough contiguous space. Segmentation works to fix this by breaking up a process into parts: Code, Data, Stack. These parts are able to be place any where in the memory space and do not have to be contiguous with each other!

12. All modern memory management schemes use paging. Briefly explain the main motivation behind paging and if paging requires hardware support.
The main motivation behind paging was to create a solution for external fragmentation. External fragmentation made it so that large processes could not be loaded into memory to be executed if there was not enough contiguous space for it to be held. This problem was solved with paging, by breaking the memory into equal sized slots called frames, and breaking the processes up into pages of the same size, so that they may always fit into non-contiguous frames. Paging does require hardware support, as it requires a MMU.

13. List 3 things that are stored in a Page Table Entry (PTE) and briefly explain what each is used for?
- Modify bit
    - The modify bit signals if the page has been modified (if the page is dirty)
- Reference bit
    - The reference bit signals if the page has been read or written to
- Frame Number
    - Which frame this page belongs to

14. List 1 advantage and 1 disadvantage of paging.
One advantage of paging is that it eliminates external fragmentation
One disadvantage is that it is memory reference intensive. By that I mean when accessing memory you had to make 2 references rather than one (page table, then memory)

15. What's the motivation behind multi-level page tables? Briefly explain.
The motivation behind multi-level page tables is to save memory used by page tables in systems with large virtual address spaces. This is because single-level page tables must have one page table entry (PTE) for every page, even if the pages are not in use. To combat this, page tables are broken up into multiple levels, and only the parts of the page table needed for currently used addresses are kept in memory. This decreases memory use, but increases memory reference overhead.

16. A machine has 48-bit virtual addresses and 32-bit physical addresses with a page size of 8 KB.
(1) How many entries are needed for the page table?
1 KB = 1024 = 2^10 → 8 KB = 8 * 2^10 → 2^3 * 2^10 → 2^13 bits
48-bit address space → 2^48 bits

Entries for page table → Address space size / size of pages
Entries for page table → 2^48 / 2^13 → 2^35

(2) How many frames are there in the physical memory?
Frames need to be same size as pages → frame size = 8 KB → 2^13 → 8(2^10)
32-bit address space → 2^32

frames → Address space size / size of frames
frames → 2^32 / 2^13 → 2^19

17. What is the effect of allowing two entries in a page table to point to the same page frame in memory? Explain how this effect can be used to decrease the amount of time and memory needed to copy a large amount of memory from one place to another. What would the effect of updating some byte in one page be on the other page? How would you make sure that updating a byte in one page does not change the same byte in the other page?

If two entries in a page table point to the same page frame in memory, both pages refer to the same physical memory. That means any read from either page returns the same data, and any write to one page affects the other page. This decreases the amount of time and memory needed to copy a large amount of memory from one place to another because instead of actually moving the memory you can just have a new page point to the same physical address.

18. Consider a logical address space of 8 pages of 1024 bytes each, mapped onto a physical memory of 32 frames.
a. How many bits are there in the logical address?
8 pages each 1 KB → 8(2^10) → 2^3(2^10) → 2^13 → 13 bits
b. How many bits are there in the physical address?
Frames space = 32 frames each frame 1 KB. 32(1^10) → 2^5(2^10) → 2^15
15 bits
c. If the OS requires all pages of process to be in memory for it to run, how many processes can the OS load to memory at the same time?
32 frames / 8 pages per process = 4

19. Consider a 32-bit computer that has 1GB of physical memory. This machine employs single-level paging with 8192 byte (8 KB) frames.
a. Show the parts of a virtual address, i.e., how many bits are used for offset, how many bits for indexing the page table?

- Since page size = 2^13 bytes → we need 13 bits for the offset
- For indexing, virtual address is 32 bits, offset is 13, so 32-13 = 19 bits.

b. How many physical frames does the machine have? How many virtual pages does the machine have? How many bytes does a process page table occupy in memory?
Physical frames → 1 GB = 2^30 bytes , each frame 2^13. So 2^30/2^13 → 2^17
Virtual → 32-bit computer = 2^32 bits, each page 2^13 → 2^32/2^13 = 2^19
Page table occupy size → # pages * size of PTE

c. A user process generates the virtual address 0x345672F3. Compute the page number and offset from the virtual address and explain how the system establishes the corresponding physical location.

Virtual Address → 0x345672F3 Page Number: 0x34567 Offset: 0x2F3

To get the corresponding physical location, the system goes to the page that corresponds to the page number which will provide the frame number. Then the system will concatente the offset to the frame number to get the physical location. For example if 0x34567 contained 0x12345 then the physical address is 0x123452F3!

d. Assume that memory access takes 100 microseconds (100,000 nanoseconds). This machine uses TLB and TLB access takes 100 nano seconds. Assume a 90% TLB hit ration, compute the effective memory access time.

EAT = (β + ξ)α + ((n+1)β + ξ)(1 - α) → (100,100).90 + (200,100).10
α → hit ratio → .90
ξ → TLB access time → 100 nanoseconds
β → Memory access time → 100,000 nanoseconds
n → # levels → 1

20. Consider a 16-bit architecture that uses single-level paging. Assume the frame size is 256 bytes, and the total physical memory size is 16KB.

a. What's the total virtual memory size? How many frames does the physical memory have?
256 bytes → 2^8 | physical memory → 16(1 KB) → 2^4(2^10) → 2^14

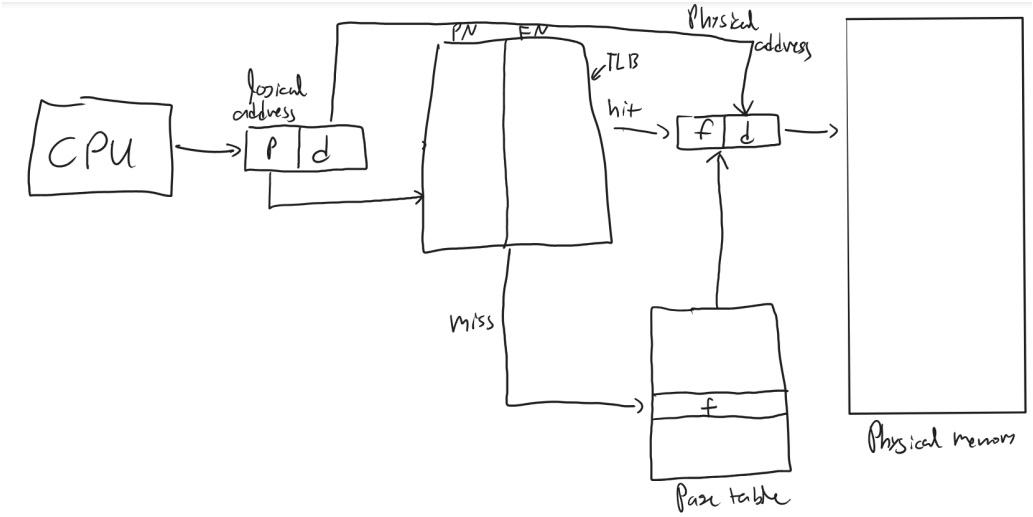
Virtual memory size → 2^16 bytes
Frame # → 2^14 / 2^8 = 2^6

b. Draw the virtual address and show how many bits are used for offset and how many bits are used for page number.
Since page size is 2^8 bytes, we need 8 bits to contain the offset.
Since we are in a 16-bit architecture, 16-8 = 8 bits so 8 bits are left for the page number

c. How many entries does a process's page table have? What's the size of a process's page table?

A process's page table is 2^16 bytes, and a pages size is 256 KB (2^8 bytes). So the processes page table can have 2^16/2^8 → 2^8 = 256 pages!

d. Show a diagram that shows how address translation occurs. Draw TLB, page table and physical memory. Assume all pages of a process are in the memory for the process to run.



e. Assume TLB search takes 1ns and memory access takes 100ns. For a TLB hit rate of 90%, compute the effective memory access time.

ETA = (B + E)a + (2B + E)(1 - a) → (100 + 1).90 + (200 + 1)(.10) → 101*.90 + 201*.10
a → hit ratio → .90
B → memory access time → 100 ns
E → TLB access time → 1 ns

21. We discussed in class that typically OSs keep a separate page table for each process. Since the logical address space of all processes are the same, the TLB must be flushed during a context switch. Suggest a fix that would NOT require the TLB to be flushed during a context switch. What are the implications of your suggestion?

A fix that makes it so that the TLB does not have to be flushed during a context switch is using Context Identifiers! This makes it so that you can take TLB entries with which process they belong to. With this, translations from other processes can remain in the TLB but are ignored!

22. The percentage of time that any page number is found in the TLB is called the hit ratio. Assume a TLB hit ratio of 90% and a single-level page table. If it takes 10 nanoseconds to lookup the TLB, and 100 nanoseconds to access memory, calculate the effective memory-access time.

EMAT = (B + E)a + (3B + E)(1 - a) → (110).90 + (310).10
a → hit ratio → .90
B → memory access time → 100 ns
E → TLB access time → 10

23. Suppose your machine has 32 bit virtual addresses and uses a 2-level page table. Virtual addresses are split into a 9 bit top-level page table field, an 11 bit second level page table field, and an offset. How large are the pages? How many pages are there in the virtual address space? If the machine has 1GB physical memory, how many frames does the memory have?

Find offset:
32 bits total = 9 bits + 11 bits + offset → offset = 12 bits

Page size → 2^offset → 2^12 → 4096 byte = 4 KB

pages → top-level bits + second-level bits = 9 + 11 = 20 bits → 2^20 pages

Frame # → 2^30 / 2^12 → 2^18 frames

24. Consider a 32-bit computer that has 1GB of physical memory. This machine employs single-level paging with 8KB frames.
a. Show the parts of a virtual address, i.e., how many bits are used for offset, how many bits for indexing the page table?

b. How many physical frames does the machine have? How many bytes does a process page table occupy in memory?
c. A user process generates the virtual address 0x345612F3. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.

d. Assume that memory access takes 100 microseconds. This machine uses TLB and TLB access takes 100 nano seconds. Assume a 90% TLB hit ration, compute the effective memory access time.

25. A certain computer provides its users with a virtual address space of 216 bytes. The computer has 212 bytes of physical memory. The virtual memory is implemented by paging with 2-level page tables, and the page size is 64 bytes.
a. Show the parts of a virtual address, i.e., how many bits are used for offset, how many bits for indexing the outer page table, and how many bits for indexing the inner page table?

b. A user process generates the virtual address 0x12F3. Explain how the system establishes the corresponding physical location. Distinguish between software and hardware operations.
26. Why would it be a bad idea to use a 2-level page table (page directory and page table) for a processor that supports 64-bit memory addresses (using a 4KB page size)?

27. What's Translation Look Aside Buffer (TLB)? What is it used for? Briefly explain.
The Translation Look Aside Buffer (TLB) is used as a way to cache PTEs to store recent virtual-to-physical address translations. This makes it so that page lookup is faster if the page has already been cached. If it has not been cached then the CPU must then go to the page table to retrieve the mapping which takes more time. Once retrieved, the mapping is cached.

28. Why does an OS need to flush the TLB during a context switch? Can you propose a fix to this problem so that the TLB need not be flushed after a context switch?

The OS needs to flush the TLB during a context switch because after a context switch, the TLB will contains entries from the previous process, which does not relate to the current process. To fix this problem so that the TLB does not need to be flushed after a context switch is by adding a context identifier to each entry in the TLB. This makes it so that each entry is mapped to a specific process, so that when a context switch occurs, the new process can just look at the TLB entries that pertain to that process and no others!

29. Consider an operating system that uses paging in order to provide virtual memory capability; the paging system employs both a TLB and a single-level page table. Assume that page tables are pinned in physical memory. Draw a flow chart that describes the logic of handling a memory reference. Your chart must include the possibility of TLB hits and misses, as well as page faults. Be sure to mark which activities are accomplished by hardware, and which are accomplished by the OS’s page fault exception handler.

30. Suppose we have an OS that uses paged virtual memory and further suppose that the OS stores the page table in CPU registers. (Thus, in this problem, you may assume that looking into the page table has negligible time cost.) Assume that it takes 8 milliseconds to service a page fault if an empty page frame is available or if the victim page is not dirty, and 20 milliseconds if the victim page is dirty. Suppose further that physical memory takes 1 microsecond to access. If the page replacement algorithm selects a dirty victim page 60 percent of the time, then what is the maximum acceptable page fault rate for an effective access time of no more than 2 microseconds? Express the page fault rate "p" as the probability that a memory reference results in a page fault.

31. Consider a logical (virtual) address space of eight pages of 1024 words each, mapped onto a physical memory of 32 frames.
a. How many bits are there in the logical address space?
b. How many bits are there in the physical address space?
c. Assume that the OS requires all pages of a process to be in memory to execute. What's the maximum number of processes that can be admitted? What's the minimum number of processes that can be admitted?

32. Consider a system with an average memory access time of 100 nano-seconds, a three level page table (meta-directory, directory, and page table). For full credit, your answer must be a single number and not a formula.
a. If the system had an average page fault rate of 10-4 for any page accessed (data or page table related), and an average page fault took 1msec to service, what is the effective memory access time (assume no TLB or memory cache).

b. Now assume the system has no page faults, we are considering adding a TLB that will take 1 nano- second to lookup an address translation. What hit rate in the TLB is required to reduce the effective access time to 160ns?

c. Somebody suggests increasing the page size to improve the system's performance. List 2 advantages and 2 disadvantages of making such a decision.

33. Consider a memory management system that uses paging. The page size is 512 bytes. Assume that the memory contains 16 pages labeled 0 through 15 and the OS is resident on the first 4 pages of the memory. Assume that there is a single process in the system using up 3 pages of memory. Specifically, the process is resident on physical blocks 5, 7, and 11.

a. Show the current snapshot of the memory and the page table of the current process.

b. Assume now that a new process arrives and requires 2000 bytes of memory. Allocate space for this process, show its page table, and depict the current snapshot of the memory.

c. What's internal fragmentation as it applies to memory management. How many bytes are wasted for the newly arriving process in (b) due to internal fragmentation?

34. Consider an OS running on the following machine: (1) a two level page table (the pointer to the page directory is stored in a register) (2) a memory reference takes 100 nanoseconds
a. If there is no TLB in the system, and 0.001% of attempts to access physical memory cause a page fault that takes 10msec to service, what is the effective memory access time?

b. If we add a TLB and 75% of all page-table references are found there, and 0.001% of the remaining memory references to access physical memory cause a page fault that takes 10msec to service, what is the effective access time of user space memory.

35. Briefly explain how a page fault occurs and the steps that the memory management system takes to handle a page fault.
A page fault occurs when the CPU checks the page table to see if the page that corresponds to the logical address, and that specific page has not been loaded into main memory yet. When this happens, the OS traps the kernel and invokes the page fault handler. Here, the OS first validates the page. If invalid, the process terminates. If the page is valid, the OS uses a page replacement algorithm if there are no free frames for the page. If there is a free page then that free frame is assigned to the page. Once either is completed, the OS updates the PTE so that the page references the newly assigned frame number.
36. Consider the following sequence of virtual memory references generated by a single process in a pure paging system: 10, 11, 104, 170, 73, 309, 185, 245, 246, 434, 458, 364

a. Assuming a page size of 100 words, what is the reference string corresponding to these memory references?
For a page size of 100 words, each reference xxx maps to page number ⌊x/100⌋\lfloor x/100\rfloor⌊x/100⌋. Applying that: 0, 0, 1, 1, 0, 3, 1, 2, 2, 4, 4, 3

b. Determine the number of page faults under each of the following page replacement strategies, assuming that two page frames are available to the process: Optimal, FIFO, LRU

Optimal

0	0	1	1	0	3	1	2	2	4	4	3
0	0	0	0	0	3	3	3	3	3	3	3
		1	1	1	1	1	2	2	4	4	4
X	√	√	√	√	X	√	X	√	X	√	√

FIFO

0	0	1	1	0	3	1	2	2	4	4	3
0	0	0	0	0	3	3	3	3	4	4	4
		1	1	1	1	1	2	2	2	2	3
X	√	X	√	√	X	√	X	√	X	√	X

LRU

0	0	1	1	0	3	1	2	2	4	4	3
0	0	0	0	0	0	1	1	1	4	4	4
		1	1	1	3	3	2	2	2	2	3
X	√	X	√	√	X	X	X	√	X	√	X

37. What's Belady's anomaly? Briefly explain.

Belady's anomaly is when you supply a system with more frames but still result in higher rate of page faults, despite there being more frames. For example, one reference string may have a lower page fault ratio compared to the same reference string in a 4 frame environment!

38. Given the reference string 1, 2, 2, 3, 1, 2, 1, 3, 4, 3, 5, 6, 4, 3 and a physical memory of 3 frames, show the state of the memory with Optimal, FIFO, LRU page replacement algorithms. Count the total number of page faults.

Optimal

1	2	2	3	1	2	1	3	4	3	5	6	4	3
1	1	1	1	1	1	1	1	4	4	4	4	4	4
	2	2	2	2	2	2	2	2	2	5	6	6	6
			3	3	3	3	3	3	3	3	3	3	3
X	X	√	X	√	√	√	√	X	√	X	X	√	√

FIFO

1	2	2	3	1	2	1	3	4	3	5	6	4	3
1	1	1	1	1	1	1	1	4	4	4	4	4	3
	2	2	2	2	2	2	2	2	2	5	5	5	5
			3	3	3	3	3	3	3	3	6	6	6
X	X	√	X	√	√	√	√	X	√	X	X	√	X

LRU

1	2	2	3	1	2	1	3	4	3	5	6	4	3
1	1	1	1	1	1	1	1	1	1	5	5	5	3
	2	2	2	2	2	2	2	4	4	4	6	6	6
			3	3	3	3	3	3	3	3	3	4	4
X	X	√	X	√	√	√	√	X	√	X	X	X	X

39. Given the reference string 1,2,4,3,2,4,3,5,4,3,5,6,1,3,5 and a physical memory of 3 frames, show the state of the memory after each reference with Optimal, FIFO, LRU page replacement algorithms. Count the total number of page faults.

Optimal

1	2	4	3	2	4	3	5	4	3	5	6	1	3	5
1	1	1	3	3	3	3	3	3	3	3	3	3	3	3
	2	2	2	2	2	2	5	5	5	5	5	5	5	5
		4	4	4	4	4	4	4	4	4	6	1	1	1
X	X	X	X	√	√	√	X	√	√	√	X	X	√	√

FIFO

1	2	4	3	2	4	3	5	4	3	5	6	1	3	5
1	1	1	3	3	3	3	3	3	3	3	3	1	1	1
	2	2	2	2	2	2	5	5	5	5	5	5	3	3
		4	4	4	4	4	4	4	4	4	6	6	6	5
X	X	X	X	√	√	√	X	√	√	√	X	X	X	X

LRU

1	2	4	3	2	4	3	5	4	3	5	6	1	3	5
1	1	1	3	3	3	3	3	3	3	3	3	1	1	1
	2	2	2	2	2	2	5	5	5	5	5	5	3	3
		4	4	4	4	4	4	4	4	4	6	6	6	5
X	X	X	X	√	√	√	X	√	√	√	X	X	X	X

40. Consider the following page reference string: 1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 6, 5, 3, 2, 2, 1. How many page faults would occur for Optimal, FIFO, LRU page replacement algorithms assuming 4 memory frames. Remember that all frames are initially empty, so your first unique pages will all cost one fault each. Also show the state of the memory after each page reference and indicate the victim page when a page fault occurs.

41. Consider a reference string 1, 2, 3, 4, 2, 5, 7, 2, 3, 2, 1, 7, 8. How many page faults would occur for Optimal, FIFO, LRU page replacement algorithms assuming 4 memory frames? Remember all frames are initially empty, so your first unique pages will all cost one fault each.

42. Consider the following page reference string: 1, 2, 3, 4, 2, 1, 2, 5, 7, 6, 3, 2, 1, 2, 3, 4. How many page faults would occur for Optimal, FIFO, LRU page replacement algorithms assuming 4 memory frames? Remember all frames are initially empty, so your first unique pages will all cost one fault each.

43. Consider the following page reference string: 1, 3, 2, 5, 1, 3, 4, 1, 3, 2, 5. How many page faults would occur for Optimal, FIFO, LRU page replacement algorithms assuming 3 memory frames? Remember all frames are initially empty, so your first unique pages will all cost one fault each.

44. Consider the reference string 1 2 3 4 5 2 3 4 3 2 4 4 2 4 4 4. Assuming the working set strategy, determine the minimum window size such that the string generates at most 5 page faults. Show which pages are in the working set (and, thus, in memory) at each reference.

45. Why is it difficult to perfectly implement the LRU page replacement algorithm? What bit(s) in the PTE do OSs use to approximate LRU?
It is difficult to perfectly implement the LRU page replacement algorithm because any time a page is referenced and it is in the cache, you need to update its reference bit, and when a new page is referenced you need to figure out which was used least recently using its reference bit and an external data structure like a list to figure out which was used least recently.

46. Briefly explain how LRU clock or second chance algorithm works.

47. Briefly explain how LRU enhanced second chance algorithm works.

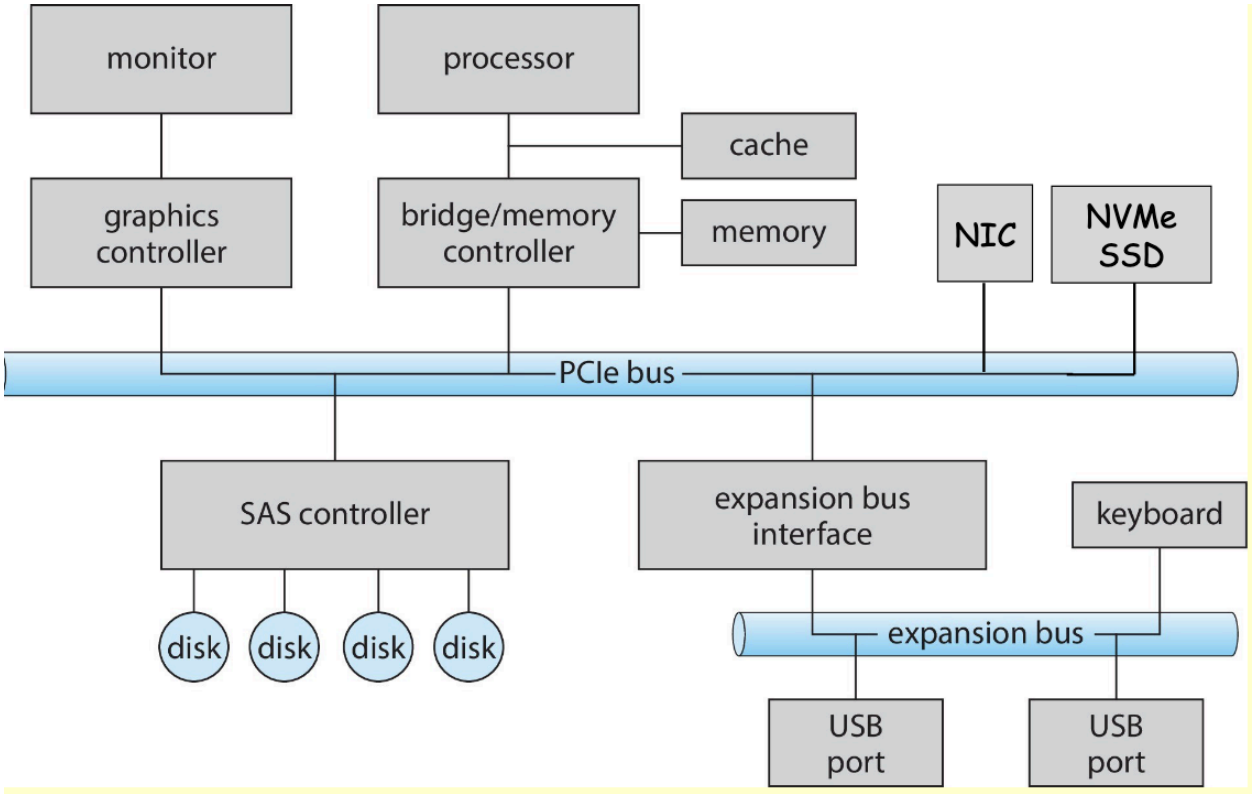
48. What's meant by the working set of a process. Given the reference string 1, 2, 3, 2, 2, 1, 3, 2, 2, 1 and a reference window size of 5, what's the process's working set at the end of the last page reference?

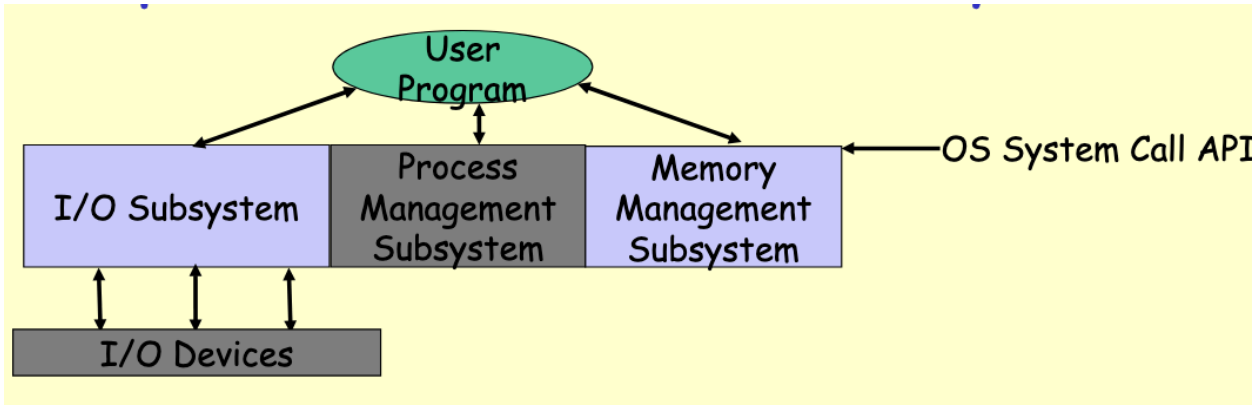
Topic 8

One of the main jobs of a computer is Input/Output (I/O)

- Get user input from keyboard – Keyboard
- Display information on the screen – Display
- Read/Save a file from/to the disk – Hard disk, SSD, CD-ROM
- Download a Web page using the browser – Network Interface Card (NIC)
- OS must control all I/O devices

Typical Architecture



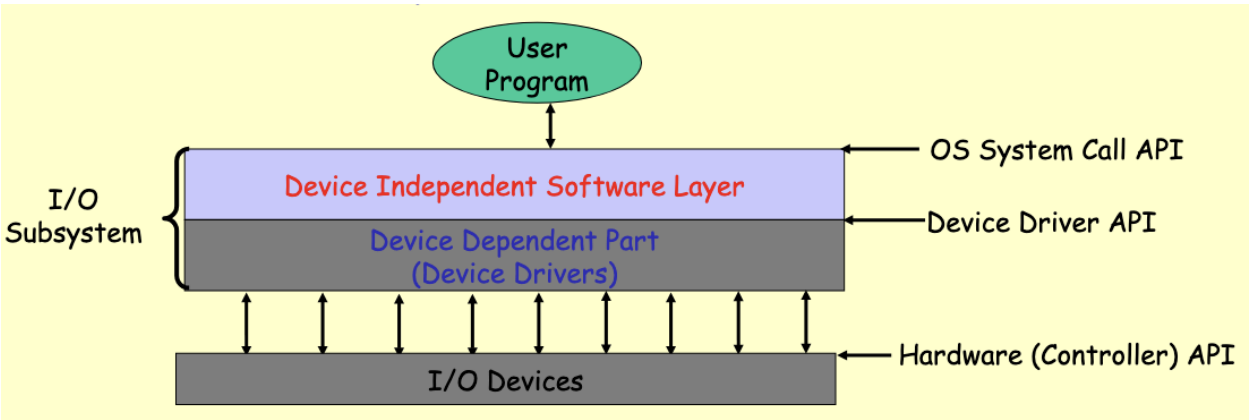


I/O Subsystem Functions

1. Provide OS-I/O Device Interaction
    a. Issue commands to the devices, catch interrupts, handle errors
2. Provide OS-Application Process Interaction
    a. Export an easy-to-use API to the applications
        i. Open, read, write, close, seek, ioctl, select

I/O Subsystem can roughly be divided into 2 layers

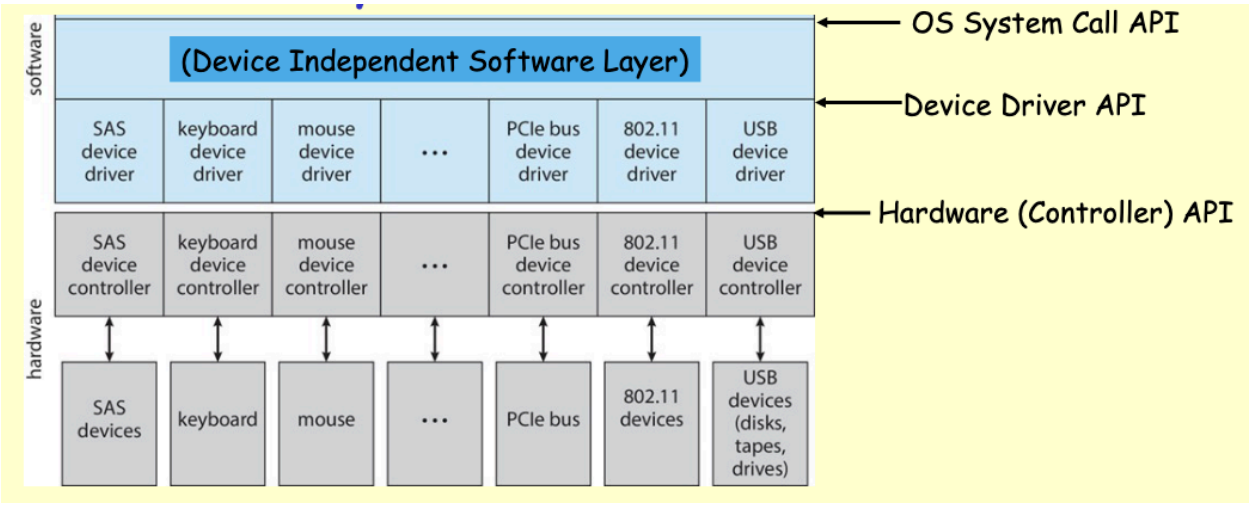
- A device independent software layer, which implements the OS-User Application Process Interface (system call API: open, read, write, close, seek, ioctl, …)
- A device dependent part, called the device driver, which implements the OS-device interaction



Device Drivers

Device drivers implement the OS-device interaction

- Interacts with the device controller using the controller interface
    - Hardware Controller API: Device registers, commands, interrupts, etc.
- Exports an interface to the Device Independent Software Layer above it
    - Device Driver API: Set of functions implemented by a device driver

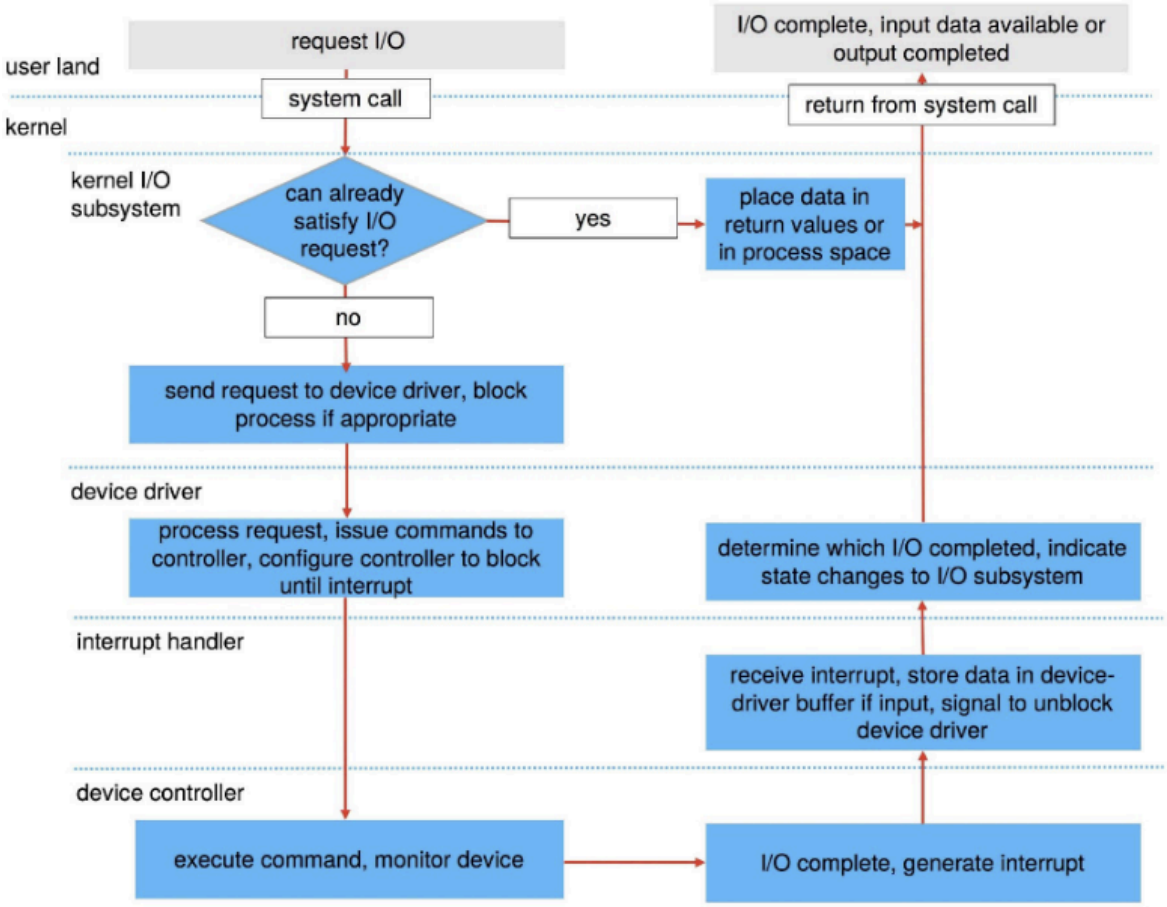


Device Independent Software Layer

Sits on top of the device drivers

- Uses the device driver API to perform actual I/O
- Exports a system call API for application processes to perform I/O from the user space

Life Cycle of an I/O Request



Device Driver API

Lots of I/O devices

- Can we define an interface for each type of device? No! Too many interfaces

Solution: Classify I/O devices into one of a small number of categories and define an interface for each group

- Character devices: keyboards, printers, mouse, …
- Block devices: Disks
- Network devices: NICs such as Ethernet cards, Wi-Fi adapters

Since each OS defines its own device driver API, a new device driver must be implemented for each OS

System Call API: Device Naming/Protection

How does an application name an I/O device such as a disk or keyboard?

- Assign symbolic names for each device and have DIS map symbolic names onto proper driver
- Symbolic names uniquely identify the device within the OS
- Setting access rights to different users?
    - Simply set the appropriate file protection rights

System Call API: I/O Call Semantics

Blocking (Synchronous) I/O Calls

- Process suspended until I/O completed
    - Easy to use and understand
    - Insufficient for some needs

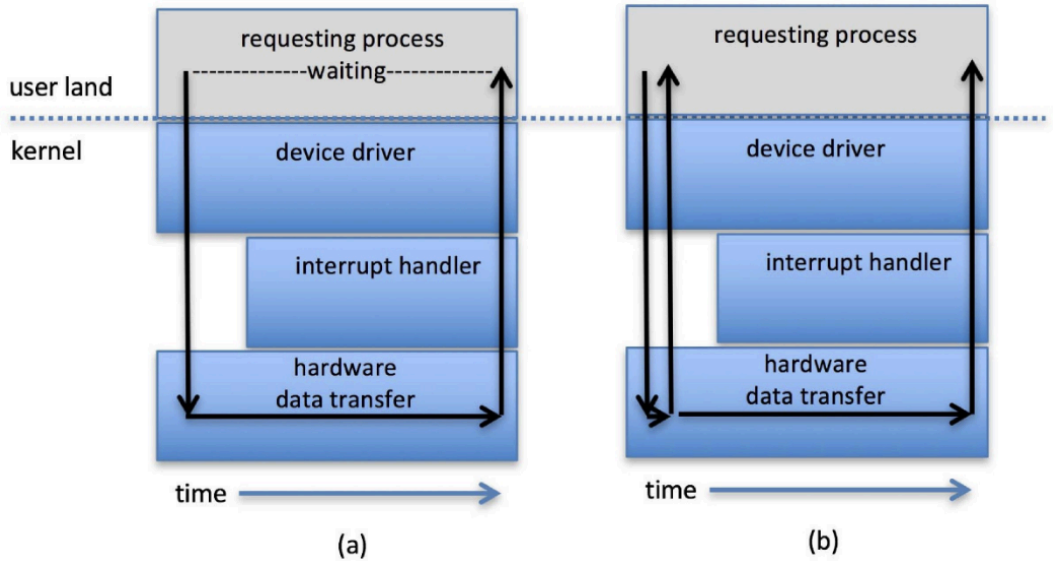
Non-Blocking I/O Calls (enabled by setting O_NONBLOCK flag)

- I/O call returns as much as available
    - Returns immediately with count of bytes read or written
    - select() to find if data ready then read() or write() to transfer

Asynchronous I/O Calls

- Process runs while I/O executes
    - I/O subsystem signals process when I/O completed (via a callback)
    - Difficult to use

System Call API: Synchronous vs Asynchronous I/O



Mode	Behavior	Use Case
Blocking	Process waits until the I/O operation completes	Simple applications where waiting is acceptable
Non-Blocking	Process continues execution; I/O call returns immediately (may fail)	Event-driven programs, network servers
Asynchronous	Process continues execution; kernel notifies when I/O completes	High-performance applications needing overlapping I/O and computation

Device Independent Software Layer Services

- Scheduling
    - Some I/O request ordering via per-device queue
        - Disk I/O scheduling
- Buffering - store data in memory while transferring between devices
    - E.g., Network packets arriving for a process asynchronously
    - Size matching: Application writes 10 bytes. Disk write is 1 block
- Caching - fast memory holding copy of data
    - Key to performance, keep the recently accessed disk blocks for faster read/write operations - called the disk I/O cache