Operating System

Yuqiao Meng

2021–9–124

Contents

1 Overview

1.1 What?

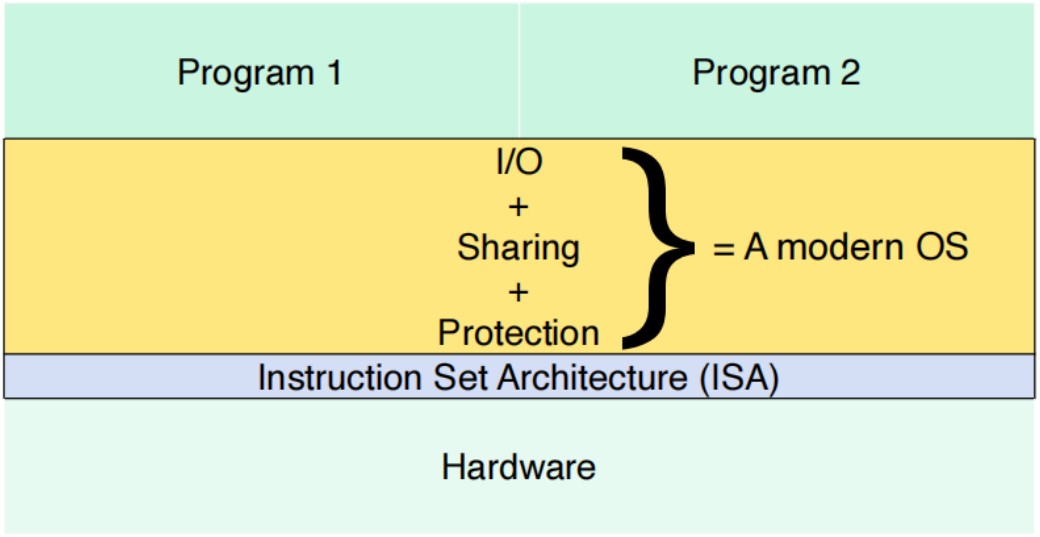
What is an Operating System? What's its reponsibility?

- A bunch of software and data residing somewhere in memory.
- The most privileged software in a computer. It can do special things, like write to disk, talk over the network, control memory and CPU usage, etc
- Manages all system resources, including CPU, Memory, and I/O devices.

1.2 Why?

Why do we need an OS?

- OS helps program to control hardwares.
- OS determines the way programs share resources.
- OS protects hardwares and programs from getting attacked.
- OS stores files persistently.



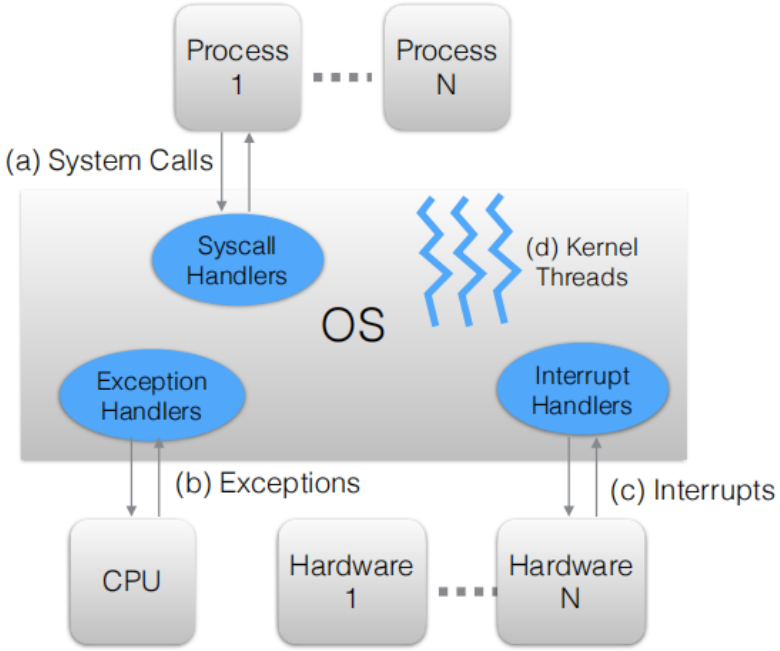
1.3 How

1.3.1 Virtualization

- Definition: OS takes a physical resource (such a sthe processor, or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use virtual form of itself.
- Resource Virtualization
  - Many(virtual)-to-one(physical): Virtual Machine
  - One-to-many: Memory Virtualization
  - Many-to-many: CPU Virtualization

1.3.2 How to invoke OS code?

- System calls: Function calls into the OS, that OS provides these calls to run programs, access memory and devices, and other related actions.
- Exceptions: CPU will raise an exception to the OS when the running program do something wrong
- Interrupts: Hardwares send interrupts to invoke OS
- Kernel Threads: Programs run in the kernel context, executing kernel level functions.

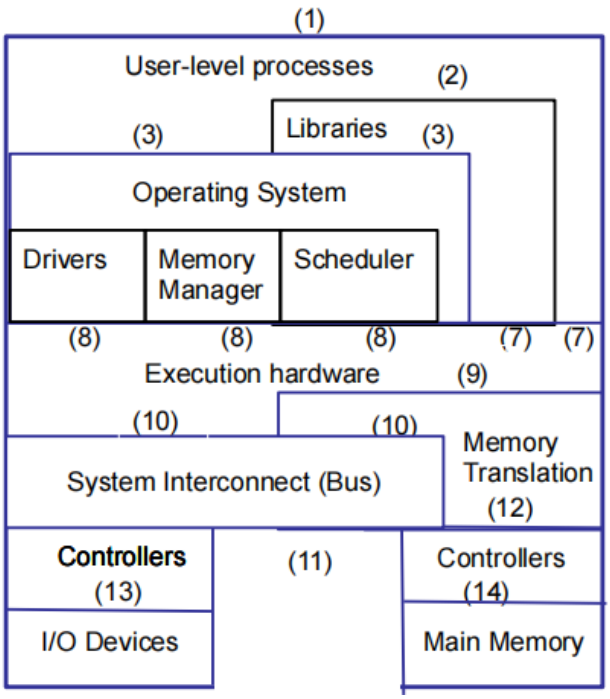


1.4 Interface

1.4.1 Explaination

- Instruction Set Architecture(ISA): the language CPU understand
- User ISA: ISA that any program can execute, it's accessible for all programs, doesn't need the service of operating system
- System ISA: ISA that only operating system is allowed to execute.
- Application Binary Interface(ABI): the combination of syscalls and User ISA(3, 7), it's the view of the world, seen by programs. It's the reason why a program complied on one OS cannot be just moved to another OS.
- Application Programmers' Interface(API): the combination of libraries and User ISA(2, 7), it's the tools programmer use to write codes.

1.4.2 Interfaces in a Computer System



- User ISA: 7
- System ISA: 8
- Syscalls: 3
- Application Binary Interface: 3, 7
- Application Programmers' Interface: 2, 7

1.5 History

- First Computer: Atanasoff–Berry computer, or ABC. ENIAC
- First OS: GM-NAA I/O, produced in 1956 by General Motors' Research division for its IBM 704.
- First language: Plankalkül, developed by Konrad Zuse for the Z3 between 1943 and 1945. Or FORTRAN

- First programmer: Ada Lovelace

2 Processes

2.1 Process

2.1.1 What?

What is a process?

- A process is a program in execution. A program is a set of instructions somewhere (like the disk).
- Once created, a process continuously does the following:
  - **Fetches** an instruction from memory.
  - **Decodes** it. i.e., figures out which instruction this is.
  - **Executes** it. it does the thing that it is supposed to do, like add two numbers together, access memory, check a condition, jump to a function, and so forth.

2.1.2 Process versus Program

How is a process different from a program?

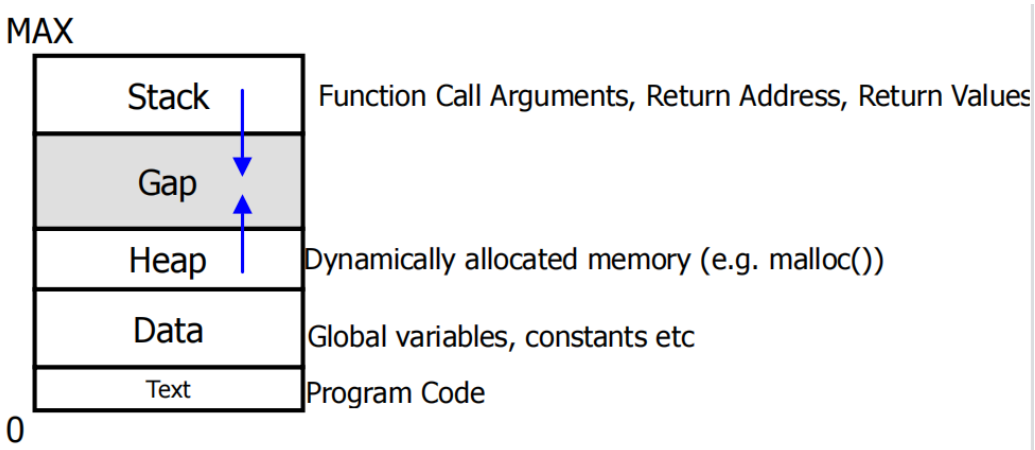
- Program: A passive entity stored in the disk, has static code and static data.
- Process: Actively executing code and the associated static and dynamic data.
- Program is just one component of a process.
- There can be multiple process instances of the same program

2.1.3 Constitution

- Memory space
- Procedure call stack

- Registers and counters
- Open files, connections
- And more.

2.1.4 Memory layout



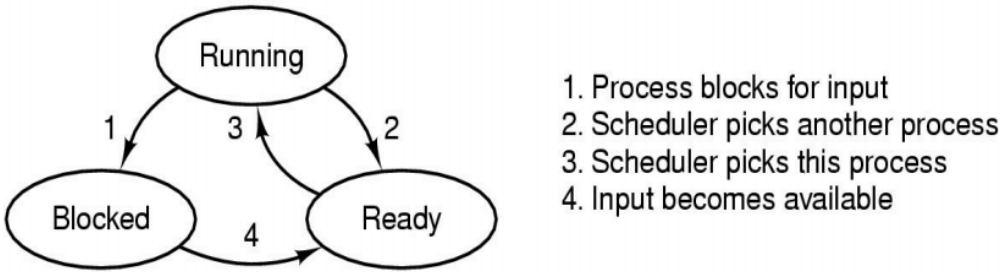
In this picture, Stack and Heap grow toward each other, that's because every process has a limited amount of space, thus let heap and stack grow toward each other from two direction can make the best use of space.

2.2 System calls

- fork(): create new process. **called once but return twice**. Usage:
  - User runs a program at command line
  - OS creates a process to provide a service: Check the directory /etc/init.d/ on Linux for scripts that start off different services at boot time.
  - One process starts another process: For example in servers
- exec(): execute a file. **No return if Success. Replaces the process' memory with a new program image. All I/O descriptors open before exec stay open after exec**.
- wait()/waitpid(): wait for child process.

- exit(): terminate a process

2.3 Lifecycle



- Ready (runnable; temporarily stopped to let another process run)
  - Process is ready to execute, but not yet executing
  - Its waiting in the scheduling queue for the CPU scheduler to pick it up.
- Running: (actually using the CPU at that instant)
- Blocked (unable to run until some external event happens).
  - Process is waiting (sleeping) for some event to occur.
  - Once the event occurs, process will be woken up, and placed on the scheduling queue

1. Running → Blocked: Occurs when the operating system discovers that a process cannot continue right now.
2. Running → Ready: Occurs when the scheduler decides that the running process has run long enough and it is time to let another process have some CPU time.
3. Ready → Running: Occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again.
4. Blocked → Ready: Occurs when the external event for which a process was waiting (such as the arrival of some input) happens

2.4 Special Process

- Orphan process
  - When a parent process dies, child process becomes an orphan process
  - The init process (pid = 1) becomes the parent of the orphan processes
- Zombie process
  - When a child dies, a SIGCHLD signal is sent to the OS, If parent doesn't wait()on the child, and child exit()s, it becomes a zombie.
  - Zombies hang around till parent calls wait() or waitpid().
  - Zombies take up no system resources, it's just a integer status kept in the OS.
  - Ways to prevent a child process from becoming a zombie:
    * Parent call wait()/waitpid() before child process exit()
    * Child parent sleep() before exit() until parent process give it a message.
    * Set act.sa_flags is SA_NOCLDWAIT

2.5 Cold-start Penalty

2.6 Context Switch

3 Inter-Process Communication

3.1 Overview

Inter-Process Communication mechanisms

- Pipe: A directional communication mechanism
- Signals: Event notification from one process to another
- Shared momery: Common piece of read/write memory, needs authorization for access

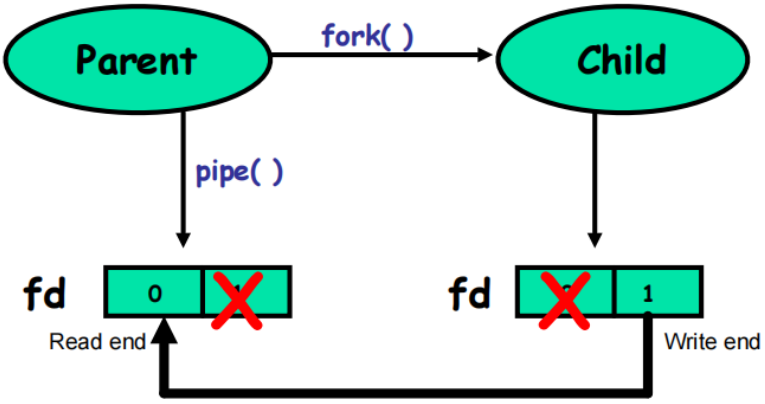
- Parent-child: Command-line arguments, including waitpid(), wait(), exit()
- Reading/modifying common files
- Semaphores: Locking and event signaling mechanism between processes
- Sockets: Not just across the network, but also between processes

3.2 Pipe

3.2.1 Abstraction

Write to one end, read from another.

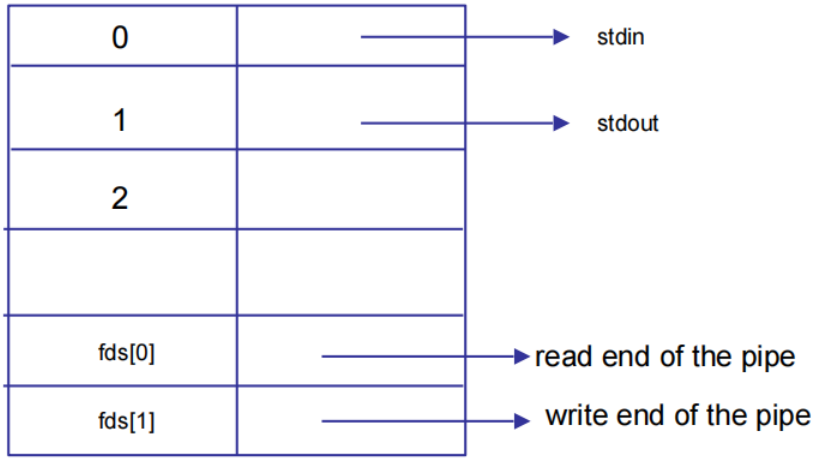
3.2.2 Parent-Child Communication Using Pipe



3.2.3 File-Descriptor Table

- Each process has a file-descriptor table
- One entry for each open file

- File includes regular file, stdin, stdout, pipes, etc.



3.2.4 Redirect Std to a File

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fds[2];
    char buf[30];
    pid_t pid1, pid2, pid;
    int status, i;

    /* create a pipe */
    if (pipe(fds) == -1) {
        perror("pipe");
        exit(1);
    }

    /* fork first child */
    if ( (pid1 = fork()) < 0) {
```

```
    perror("fork");
    exit(1);
 }

 if ( pid1 == 0 ) {
    close(1); /* close normal stdout (fd = 1) */
    dup2(fds[1], 1); /* make stdout same as fds[1] */
    close(fds[0]); /* we don't need the read end -- fds[0] */

    if( execlp("ps", "ps", "-elf", (char *) 0) < 0) {
        perror("Child");
        exit(0);
    }

/* control never reaches here */
 }

 /* fork second child */
 if ( (pid2 = fork()) < 0) {
perror("fork");
exit(1);
 }

 if ( pid2 == 0 ) {
    close(0);  /* close normal stdin (fd = 0)*/
    dup2(fds[0],0); /* make stdin same as fds[0] */
    close(fds[1]); /* we don't need the write end -- fds[1]*/

    if( execlp("less", "less", (char *) 0) < 0) {
        perror("Child");
        exit(0);
    }

/* control never reaches here */
 }

 /* parent doesn't need fds - MUST close - WHY? */
 /* The reading side is supposed to learn that the writer has
    finished if it notices an EOF condition. This can only
    happen if all writing sides are closed.*/
```

```
    /* close its reading end (for not wasting FDs and for proper
       detection of dying reader)*/
    /* close its writing end (in order to be possible to detect the
       EOF condition).*/
    close(fds[0]);
    close(fds[1]);

    /* parent waits for children to complete */
    for( i=0; i<2; i++) {
   pid = wait(&status);
   printf("Parent: Child %d completed with status %d\n", pid,
      status);
    }

}
```

3.2.5 Handling Chain of Filters Using Pipe

command 1 | *command 2* | . . . | *command N*

- First command?
  - Yes: continue
  - No: redirect stdin to previous pipe
- Last command?
  - Yes: Output
  - No:
    - create next pipe (if needed)
    - redirect stdout to next pipe
    - fork a child for next level of recursion with one command less as input
- exec the command for the current level

3.2.6 Byte-stream versus Message

- Byte-Stream abstraction: Pipe
  - Can read and write at arbitrary byte boundaries
  - Don't need to return explicit bytes of data. *read(fds[0], buf, 6)*
    - read() could reach end of input stream (EOF).
    - Other endpoint may abruptly close the connection
    - read() could return on a signal
- Message abstraction: Provides explicit message boundaries.

3.2.7 Error Handling

We must incorporate error handling with every I/O call (actually with any system call)

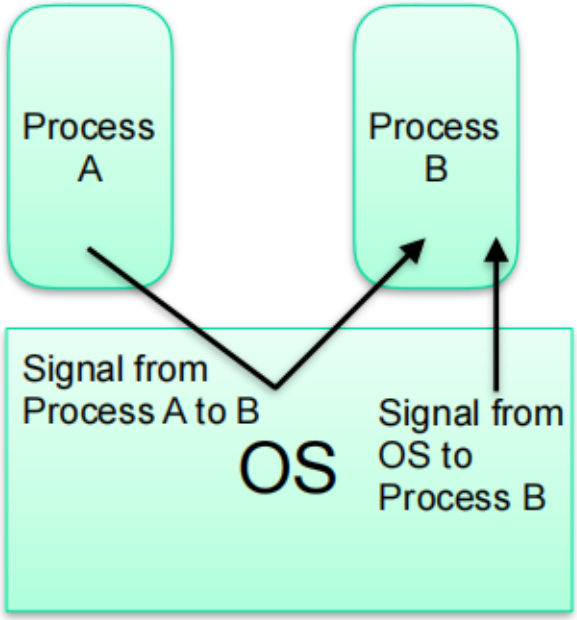
- First check the return value of every read(...)/write(...) system call
- Then
  - Wait to read/write more data
  - Handle any error conditions

```
More convinient to write a wrapper function
/* Write "n" bytes to a descriptor. */
ssize_t  writen(int fd, const void *vptr, size_t n)
{
      size_t        nleft;
      size_t       nwritten;
      const char     *ptr;

      ptr = vptr;
      nleft = n;
      while (nleft > 0) {
          if ((nwritten = write(fd, ptr, nleft))<=0){
              if (errno == EINTR)
                 nwritten = 0;  /* call write() again*/
              else return(-1);  /* error */
           }
           nleft -= nwritten;
           ptr   += nwritten;
      }
      return(n);

}
```

3.3 Signals

3.3.1 Overview

- A notification to a process that an event has occurred, comes from OS or another process
- The type of event determined by the type of signal



3.3.2 Handling Signals

- Signals can be **caught** – i.e. an action (or handler) can be associated with them
- Actions can be customized using **sigaction()**, which associates a signal handler with the signal
- **Default** action for most signals is to terminate the process. Except SIGCHLD and SIGURG are ignored by default
- Unwanted signals can be **ignored**, except SIGKILL or SIGSTOP

3.3.3 SIGCHLD

- Sent to parent when a child process terminates or stops
- If act.sa_handler is SIG_IGN, SIGCHLD will be ignored (default behavior)
- If act.sa_flags is SA_NOCLDSTOP, SIGCHLD won't be generated when children stop

- If act.sa_flags is SA_NOCLDWAIT, children of the calling process will not be transformed into zombies when they terminate
- These need to be set in sigaction() before parent calls fork()

Usage: handling child's exit without blocking on wait()

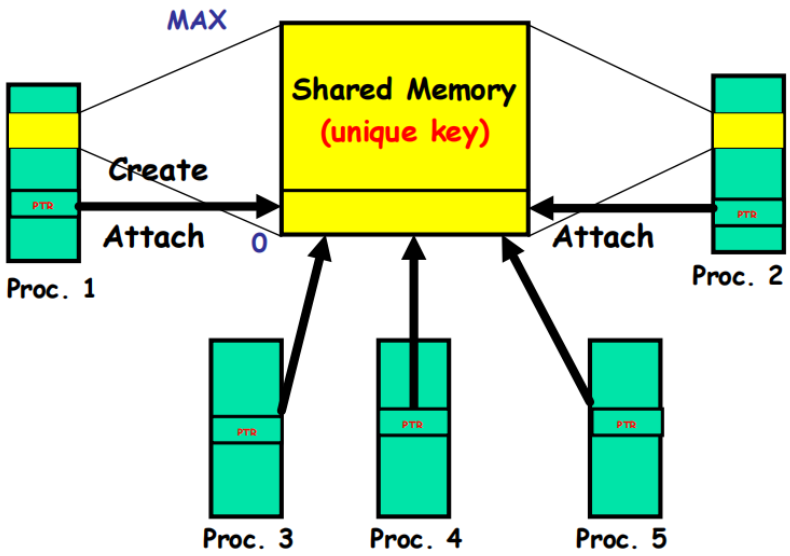
- Parent could install a signal handler for SIGCHLD
- Call wait(...)/waitpid(...)inside the signal handler

```
// sigchild.c
void handle_sigchld(int signo) {
    pid_t pid;
    int stat;

    pid = wait(&stat); //returns without blocking
    printf("child process exits.");
}
```

3.4 Shared Memory

3.4.1 Overview

Common chunk of read/write memory among processes.



3.4.2 Creating

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024 /* make it a 1K shared memory segment */

int main(void)
{

       key_t key;
       int shmid;
       char *data;
       int mode;

       /* make the key: */
       /* The ftok() function uses the identity of the file named
          by the given pathname (which must refer to an existing,
          accessible file) and the least significant 8 bits of
          proj_id (which must be nonzero) to generate a key_t type
          System V IPC key, suitable for use with msgget(2),
          semget(2), or shmget(2). */
       /* The resulting value is the same for all pathnames that
          name the same file, when the same value of proj_id is
          used. The value returned should be different when the
          (simultaneously existing) files or the project IDs
          differ.*/
       if ((key = ftok("test_shm", 'X')) < 0) {
            perror("ftok");
            exit(1);
       }

       /* create the shared memory segment: */
       /* shmget() returns the identifier of the System V shared
          memory segment associated with the value of the argument
          key. It may be used either to obtain the identifier of a
```

```
        previously created shared memory segment (when shmflg is
        zero and key does not have the value IPC_PRIVATE), or to
        create a new set.*/
    /* A new shared memory segment, with size equal to the value
        of size rounded up to a multiple of PAGE_SIZE, is created
        if key has the value IPC_PRIVATE or key isn't
        IPC_PRIVATE, no shared memory segment corresponding to
        key exists, and IPC_CREAT is specified in shmflg.*/
    /* If shmflg specifies both IPC_CREAT and IPC_EXCL and a
        shared memory segment already exists for key, then
        shmget() fails with errno set to EEXIST. (This is
        analogous to the effect of the combination O_CREAT |
        O_EXCL for open(2).) */
    /* 0644 means permissions of owner, group and user */
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT |
        IPC_EXCL )) < 0) {
         perror("shmget");
         exit(1);
    }

    return(0);
}
```

3.4.3 Attach and Detach

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024 /* make it a 1K shared memory segment */

int main(int argc, char *argv[])
{

    key_t key;
    int shmid;
```

```
char *data;
int mode;


/* make the key: */
if ((key = ftok("test_shm", 'X')) == -1) {
     perror("ftok");
     exit(1);
}


/* connect to the segment. */
/* There's no IPC_CREATE. Because if there was one this
   function would create a new shared memory.*/
if ((shmid = shmget(key, SHM_SIZE, 0644)) == -1) {
     perror("shmget");
     exit(1);
}


/* attach to the segment to get a pointer to it: */
 /* The shmat() function attaches the shared memory segment
     associated with the shared memory identifier specified
     by shmid to the address space of the calling process.*/
 data = shmat(shmid, (void *)0, 0);
 if (data == (char *)(-1)) {
     perror("shmat");
     exit(1);
 }


 /* read or modify the segment, based on the command line: */
 if (argc == 2) {
     printf("writing to segment: \"%s\"\n", argv[1]);
     strncpy(data, argv[1], SHM_SIZE);
 } else
     printf("segment contains: \"%s\"\n", data);


 /* detach from the segment: */
```

```
        if (shmdt(data) == -1) {
            perror("shmdt");
            exit(1);
        }

       return(0);
}
```

3.4.4 Deleting

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024 /* make it a 1K shared memory segment */

int main(void)
{
       key_t key;
       int shmid;
       char *data;
       int mode;

       /* make the key: */
       if ((key = ftok("test_shm", 'X')) == -1) {
            perror("ftok");
            exit(1);
       }

       /* connect to memory segment: */
       if ((shmid = shmget(key, SHM_SIZE, 0644)) == -1) {
            perror("shmget");
            exit(1);
       }

       /* delete he segment */
```

```
        /* IPC_RMID: Remove the shared memory identifier specified
           by shmid from the system and destroy the shared memory
           segment and shmid_ds data structure associated with it.
           IPC_RMID can only be executed by a process that has an
           effective user ID equal to either that of a process with
           appropriate privileges or to the value of shm_perm.cuid
           or shm_perm.uid in the shmid_ds data structure associated
           with shmid.*/
        if( shmctl(shmid, IPC_RMID, NULL) == -1) {
             perror("shmctl");
             exit(1);
        }

        return(0);
}
```

3.4.5 Command

- ipcs: Lists all IPC objects owned by the user
- ipcrm: Removes specific IPC object

4 Threads

4.1 Problem

Want to do multiple tasks Concurrently

- Start two processes
  - fork() is expensive
  - cold-start penalty

Processes may need to talk to each other

- Two different address spaces, so we need to use IPC
  - kernel transitions are expensive
  - May need to copy data from a user to kernel to another user
  - Inter-process Shared memory is a pain to set up

4.2 Solution

4.2.1 Event-driven programming

- Make one process do all the tasks
- Busy loop polls for events and executes tasks for each event
- No IPC needed
- **Length of the busy loop** determines response latency
- Stateful event responses complicate the code

```
while(1)
{
    Check pending events;
    if (event 1) do task 1;
    if (event 2) do task 2;
    // ...
    if (event N) do task N;
}
```

4.2.2 Threads

Multiple threads of execution per process

4.3 Threads

Shared Resource

- virtual address space(code, heap and static data)
- Open descriptors (files, sockets etc)
- Signals and Signal handlers

Non-Shared resources

- Program counter
- Stack, stack pointer

- Registers
- Thread ID
- Errno
- Priority

4.3.1 Address space layout

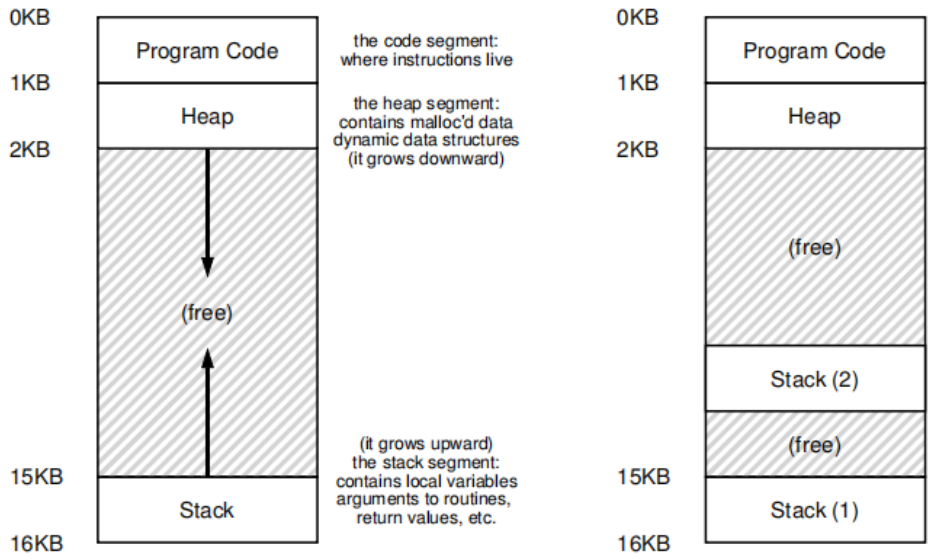


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

4.3.2 Advantages

- Lower inter-thread context switching overhead than processes
- No Inter-process communication
  - Zero data transfer cost between threads
  - Only need inter-thread synchronization
- Threads can be pre-empted at any point
  - Long-running threads are OK

  - As opposed to event-driven tasks that must be short.

- Threads can exploit parallelism, but it depends... more later
- Threads could block without blocking other threads, but it depends ... more later

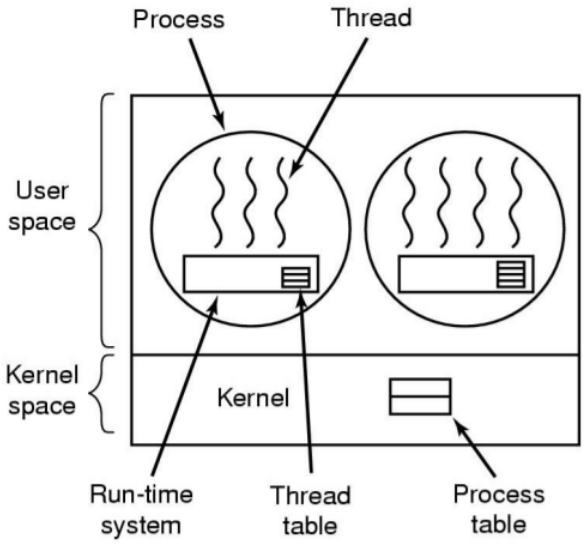
4.3.3 Disadvantages

- Shared State!
  - Global variables are shared between threads.
  - Accidental data changes can cause errors.
- Threads and signals don't mix well
  - Common signal handler for all threads in a process
  - Which thread to signal? Everybody!
  - Royal pain to program correctly.
- Lack of robustness. Crash in one thread will crash the entire process.
- Some library functions may not be thread-safe
  - Library Functions that return pointers to static internal memory. E.g. gethostbyname()
  - Less of a problem these days.

4.3.4 Types of Threads

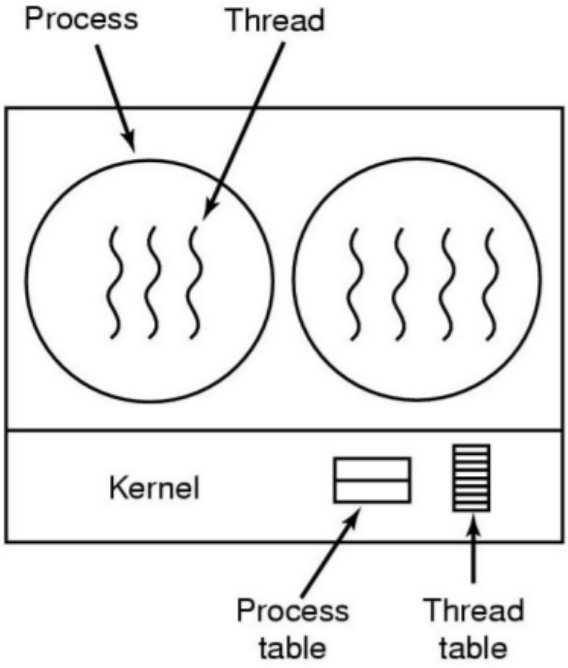
User-level threads

- User-level libraries provide multiple threads,
- OS kernel does not recognize user-level threads
- Threads execute when the process is scheduled

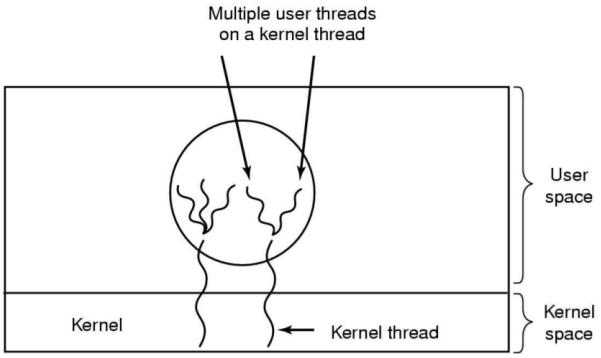


Kernel-level threads

- OS kernel provides multiple threads per process
- Each thread is scheduled independently by the kernel's CPU scheduler



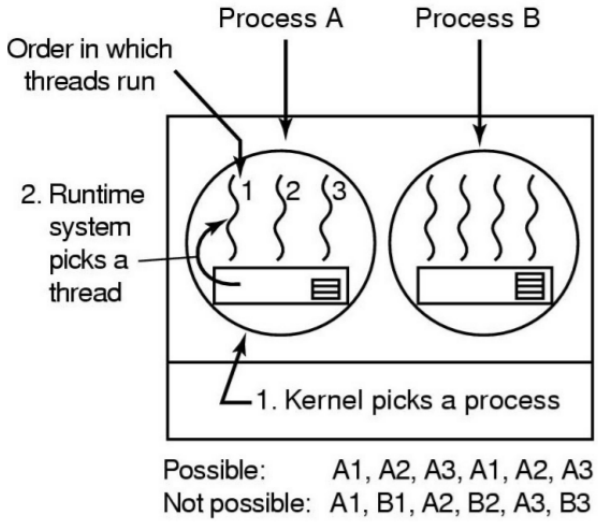
Hybrid Implementations



Multiplexing user-level threads within each kernel- level threads

4.3.5 Scheduling

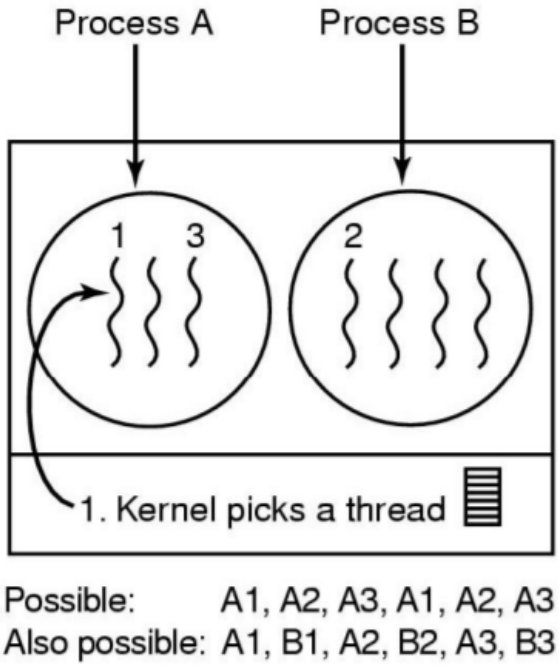
Local Thread Scheduling



- Next thread is picked from among the threads belonging to the *current process*
- Each process gets a timeslice from kernel
- Then the timeslice is *divided up* among the threads within the current process

- Local scheduling can be implemented with either Kernel-level Threads or user-level threads
- Scheduling decision requires only *local knowledge* of threads within the current process.

Global Thread Scheduling



- Next thread to be scheduled is picked up from *ANY* process in the system.
- Timeslice is allocated at the granularity of threads
- Global scheduling can be implemented only with kernel-level threads: for Picking the next thread requires global knowledge of threads in all processes.

4.3.6 Thread Creation and Termination

- Creation

```
int pthread_create( pthread_t * thread, pthread_attr_t
    * attr,
void * (*start_routine)(void *), void * arg);
```

- Two ways to perform thread termination
  - Return from initial function

```
void pthread_exit(void * status)
```

- Waiting for child thread in parent

```
pthread_join()
```

  - equivalent to waitpid

4.3.7 Code

Example

```
// shared counter to be incremented by each thread
int counter = 0;
main()
{
    pthread_t tid[N];
    for (i=0;i<N;i++) {
        /*Create a thread in thread_func routine*/
        Pthread_create(&tid[i], NULL, thread_func, NULL);
    }
    for(i=0;i<N;i++) {
        /* wait for child thread */
        Pthread_join(tid[i], NULL);
    }
    void *thread_func(void *arg)
    {
        /* unprotected code race condition */
        counter = counter + 1;
    }
    return NULL; // thread dies upon return
```

```
}
```

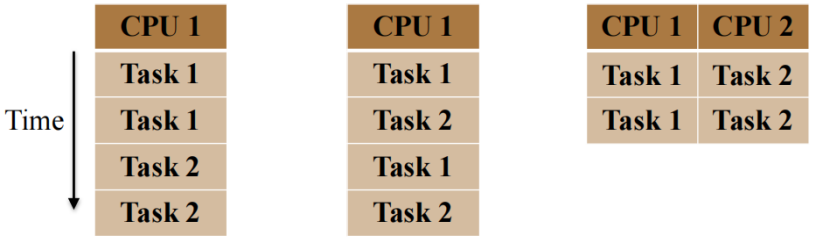
4.3.8 pthread Synchronization Operations

```
// Mutex operation
pthread_mutex_init()
pthread_mutex_lock()
pthread_mutex_unlock ()
pthread_mutex_trylock ()

// Condition variables
pthread_cond_wait ()
pthread_cond_signal ()
pthread_cond_broadcast ()
pthread_cond_timedwait ()
```

5 Concurrency

5.1 Overview

- Sequential: one after another(two tasks executed on one CPU one after another)
- Concurrent: "juggling" many things within a time window(two tasks share a single CPU over time)
- Parallel: do many things simultaneously(two threads executing on two different CPUs simultaneously)



Concurrent tasks must be either

- execute independently
- synchronize the shared resource
  - Shared memory
  - Pipes
  - Signals

5.2 Critical Section

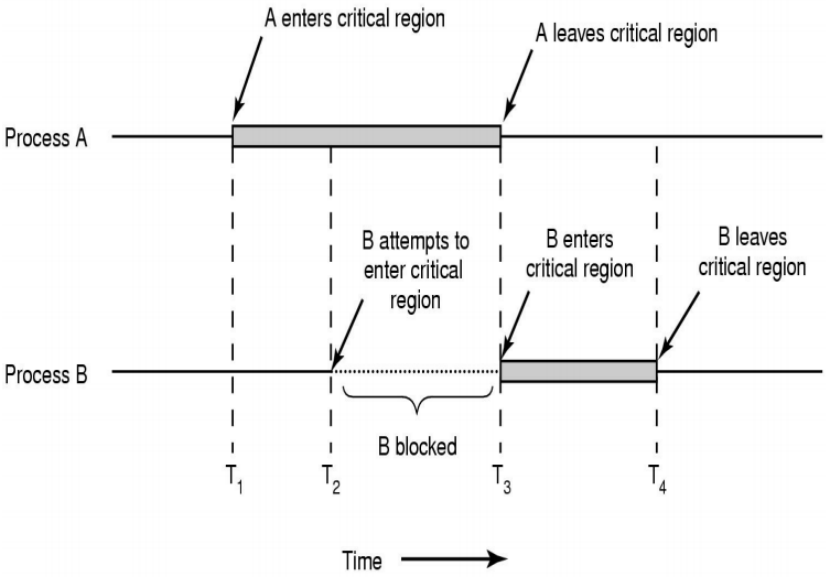
- Definition: A section of code in a concurrent task that **modifies or accesses** a resource shared with another task.
- Example: A piece of code that reads from or writes to a shared memory region

5.3 Race Condition and Deadlocks

- Race Condition: Incorrect behavior of a program due to concurrent execution of critical sections by two or more threads
- Deadlocks: When two or more processes stop making progress **indefinitely** because they are all waiting for each other to do something

5.3.1 Mutual Exclusion

Don't allow two or more processes to execute their critical sections concurrently (on the same resource)



5.3.2 Conditions for correct mutual exclusion

- No two processes are simultaneously in the critical section
- No assumptions are made about speeds or numbers of CPUs
- No process must wait forever to enter its critical section. Waiting forever indicates a *deadlock*
- No process running outside its critical region may block another process running in the critical section

The first two conditions are enforced by the operating system's implementation of locks, but the other two conditions have to be ensured by the programmer using the locks.

5.3.3 Typs of Locks

- Blocking locks
  - Give up CPU till lock becomes available

```
while(lock unavailable)
    yield CPU to others; // or block till lock available
return success;
```

  - Usage:

```
Lock(resource); // Claim a shared resource
Execute Critical Section;//access or modify the shared
    resource
Unlock(resource); // unclaim shared resource
```

  - Advantage: Simple to use. Locking always succeeds. . . ultimately
  - Disadvantage: Blocking duration may be indefinite
    * Process is moved out of "Running" state to "Blocked" state, and return to running will cost much resource
    * Delay in getting back to running state if lock becomes available soon after blocking

- Non-blocking locks
  - Don't block if lock is unavailable

```
if(lock unavailable)
    return failure;
else
    return success
```

  - Usage

```
if(TryLock(resource) == success)
    Execute Critical Section;
    Unlock(resource);
else
    Do something else; // plan B
```

  - Advantage: No unbounded blocking
  - Disadvantage: Need a "plan B" to handle locking failure

- Spin locks

- Don't block. Instead, constantly poll the lock for availability

```
while (lock is unavailable)
continue; // try again
return success;
```

- Usage: Just like blocking locks

```
SpinLock(resource);
Execute Critical Section;
SpinUnlock(resource);
```

- Advantage: Very efficient with short critical sections, if you expect a lock to be released quickly
- Disadvantage:
  * Doesn't yield the CPU and wastes CPU cycles, Bad if critical sections are long: P1 is in the ready queue, but P2 is doing spin with CPU until scheduler interrupts it and give CPU to P1
  * Efficient only if machine has multiple CPUs

5.3.4 Best practices for locking

- **Associate locks with shared resources, NOT code.**
- **Guard each shared resource by a separate lock.**
- **OS cannot enforce these properties**

5.3.5 Deadlock Solution

Deadlock can only be prevented, once it happens, programmers can't solve it by killing the process or enforcing the process to give up the lock.
Right Solution: Lock Ordering

- Sort the locks in a fixed order (say L1 followed by L2)
- Always acquire subset of locks in the sorted order.

5.3.6 Priority Inversion

Conditions

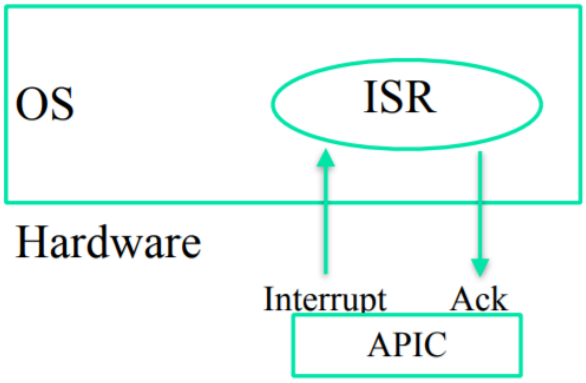
- static priority system
- synchronization between processes

Example

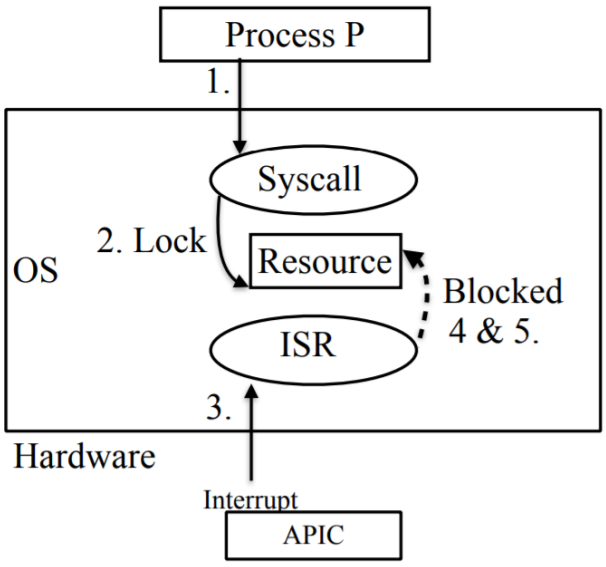
- Definition:
  - Ph – High priority
  - Pm – Medium priority
  - Pl – Low priority
- Procedure
  - Pl acquires a lock L
  - Pl starts executing critical section
  - Ph tries to acquire lock L and blocks
  - Pm becomes "ready" and preempts Pl from the CPU.
  - Pl might never exit critical section if Pm keeps preempting Pl
  - So Ph might never enter critical section
- Problem: A high priority process Ph is blocked waiting for a low priority process Pl, Pl cannot proceed because a medium priority process Pm is executing
- Solution: Priority Inheritance
  - Temporarily increase the priority of Pl to HIGH PRIORITY
  - Pl will be scheduled and will exit critical section quickly
  - Then Ph can execute

5.3.7 Interrupts and Locks

- Interrupts invoke *interrupt service routines (ISR)* in the kernel
  - ISR must process the interrupt quickly and return: because there may be some other pending interrupts waiting to be delivered.
  - So ISRs must *never block or spin on a lock*.



5.3.8 Interrupts and Deadlocks — Problem



1. P makes a syscall.

2. Syscall acquires lock
3. ISR preempts P* *But P is still in the running state*
4. ISR attempts to lock
5. ISR blocks (since lock is taken)
6. Deadlock!

5.3.9 Interrupts and Deadlocks — Solutions

1. Don't lock in ISR!: Defer any locking work to thread context (softirqs in Linux)
2. If you must lock, use *trylock*() instead of lock() in ISR
   - *trylock*() = if lock is available then get it, else return with error
   - Write code to handle unavailable lock
3. Or disable interrupts in thread T before locking
   - If ISR cannot run when lock is acquired by T, then there's no deadlock.
   - When ISR runs, it assumes that T doesn't have the lock.
   - But, disabling interrupts too long is also not a good idea.

6 Semaphores, Condition Variables, Producer Consumer Problem

6.1 Semaphores

6.1.1 Definition

Can be seen as a non-negative integer

- Semaphore is a fundamental synchronization primitive used for
  - Locking around critical regions
  - Inter-process synchronization

- A semaphore "sem" is a special integer on which only two operations can be performed
  - DOWN(sem)
  - UP(sem)

6.1.2 DOWN(sem) Operation

- If (sem > 0) then
  - Decrements sem by 1
  - The caller continues executing.
  - This is a **successful** down operation.
- If (sem $== 0$) then
  - Block the caller
  - The caller blocks until another process calls an UP.
  - The blocked process wakes up and tries DOWN again.
  - If it succeeds, then it moves to **ready** state
  - Otherwise it is blocked again till someone calls UP.
  - And so on

6.1.3 UP(sem) Operation

- This operation increments the semaphore sem by 1.
- If the original value of the semaphore was 0, then UP operation wakes up **all processes** that were sleeping on the DOWN(sem) operation.
- All woken up processes compete to perform DOWN(sem) again. Only one of them succeeds and the rest are blocked again.

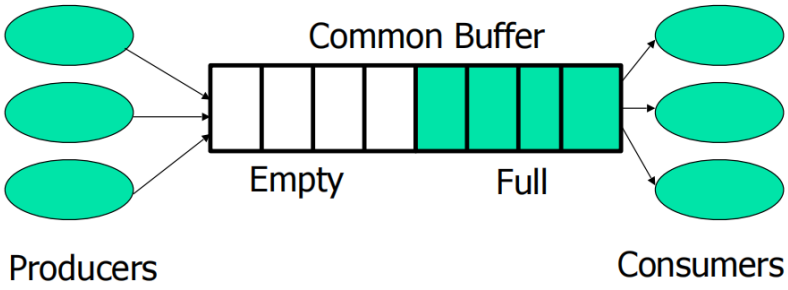
6.1.4 Mutex

A simply binary semaphore

- Used as a LOCK around critical sections
- Locking a mutex means calling Down(mutex)
- Unlocking a semaphore means calling UP(mutex)

6.2 Producer-Consumer Problem

6.2.1 Definition



- Producers and consumers run in concurrent processes.
- Producers produce data and consumers consume data.
- Producer informs consumers when data is available
- Consumer informs producers when a buffer is empty.
- Three types of synchronization needed
  - Locking the buffer to prevent concurrent modification
  - Locking the buffer to prevent concurrent modification and getting
  - Informing the other side that data/buffer is available

6.2.2 Solution

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void){
    int item;
    while(true){
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void){
    int item;
    while(true){
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

6.2.3 Using Semaphore

6.2.4 POSIX interface

- *sem_open*()
- *sem_init*()

- $sem\_wait(), sem_trywait()$
- $sem\_post()$
- $sem\_close()$
- $sem\_destroy()$
- $sem\_getvalue()$
- $sem\_unlink()$ – Ends the connection to an open semaphore and causes the semaphore to be removed when the last process closes it.

6.3 Monitors and Condition Variables

6.3.1 Definition

```
monitor example
        integer i;
        condition c;

        procedure Function1()

        .
        .           wait(c);
        .
        end;

        procedure Function2()

        .
        .           signal(c);
        .
        end;
end monitor;
```

- Monitor is a collection of critical section procedures (functions): i.e. functions that operate on shared resources
- There's **one global lock** on all procedures in the monitor. Only one procedure can be executed at any time
- **wait(c)** : releases the lock on monitor and puts the calling process to sleep. Automatically re-acquires the lock upon return from wait(c).
- **signal(c)**: wakes up all the processes sleeping on c; the woken processes then compete to obtain lock on the monitor

6.3.2 P-C problem with monitors and condition variables

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count - 1;
        if count = N - 1 then signal(full)
    end;
    count := 0;
end monitor;
```

6.4 Atomic Locking – TSL Instruction

- Instruction format: TSL Register, Lock
- Lock
  - Located in memory.
  - Has a value of 0 or 1

- Register: One of CPU registers
- TSL does the following two operations **atomically (as one step)**
  - Register := Lock; // Copy the old value of Lock to Register
  - Lock := 1; // Set the new value of Lock to 1
- TSL is a basic primitive using which other more complex locking mechanisms can be implemented.

Implementation of Mutex Using TSL

```
mutex_lock:
    TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
    CMP REGISTER,#0             | was mutex zero?
    JZE ok                      | if it was zero, mutex was unlocked, so return
    CALL thread_yield           | mutex is busy; schedule another thread
    JMP mutex_lock              | try again later
ok: RET | return to caller; critical region entered


mutex_unlock:
    MOVE MUTEX,#0               | store a 0 in mutex
    RET | return to caller
```

In C-syntax:

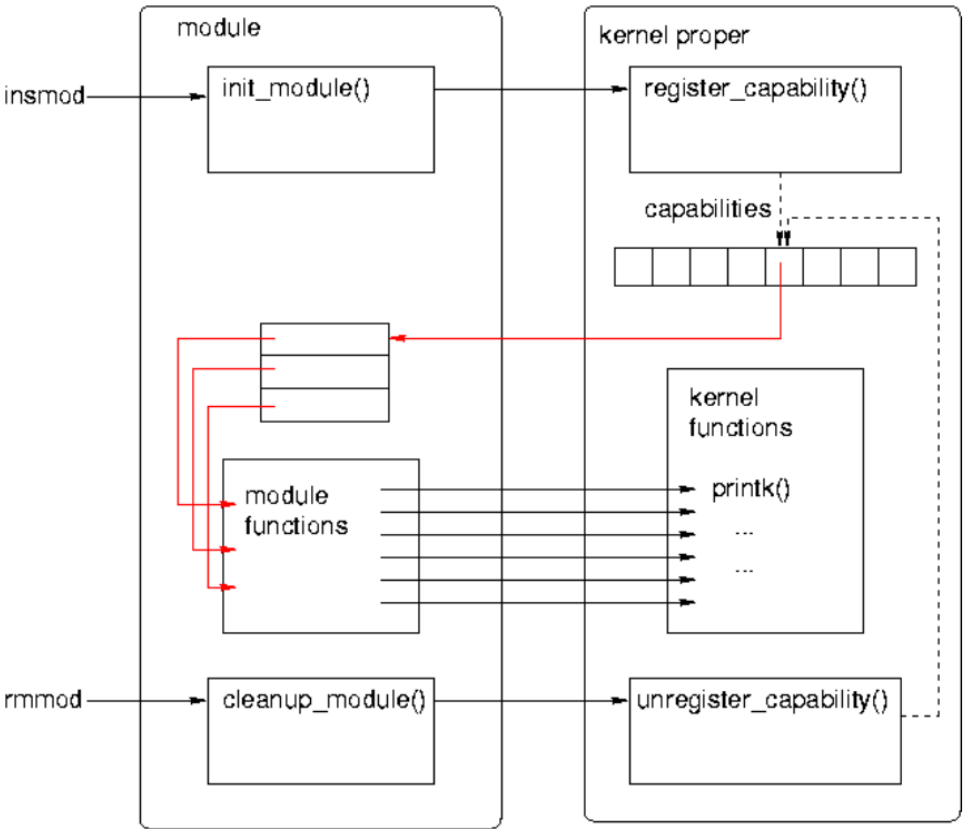
```
void Lock(boolean *lock) {
    while (test_and_set(lock) == true);
}
```

7 Kernel Modules

7.1 Definition

- Allow code to be added/removed to the kernel, dynamically
- Only those modules that are needed are loaded. Unload when no longer required - frees up memory and other resources
- Reduces kernel size.
- Enables independent development of drivers for different devices

- Any kercel module can do anything with full privilege, so it's important to truct the developer before installing a module. devices



1. init_module: do register
2. register_capability: tell the kernel where to call functions
3. OS keep pointers to the capabilities functions
4. Get invoced when needed
5. unregister_capability
6. cleanup_module

7.2 Development

7.2.1 Hello World Example

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("DUAL BSD/GPL");
// called when module is installed
int __init hello_init()
{
 printk(KERN_ALERT "mymodule: Hello World!\n");
 return 0;
}
// called when module is removed
void __exit hello_exit()
{
 printk(KERN_ALERT "mymodule: Goodbye, cruel world!!\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

7.2.2 Compile

- Makefile
  - obj-m := testmod.o
  - For multiple files: module-objs := file1.o file2.o
- Compiling: $ make -C /lib/modules/$(uname -r)/build M=‘pwd‘ modules

7.2.3 Module Utilities

- sudo insmod hello.ko
  - Inserts a module
  - Internally, makes a call to sys_init_module
  - Calls vmalloc() to allocate kernel memory
  - Copies module binary to memory

  - Resolves any kernel references (e.g. printk) via kernel symbol table
  - Calls module's initialization function
- modprobe hello.ko: Same as insmod, except that it also loads any other modules that hello.ko references
- sudo rmmod hello
  - Removes a module
  - Fails if module is still in use
- sudo lsmod
  - Tells what modules are currently loaded
  - Internally reads /proc/modules

7.2.4 Things to Remember

- Modules can call other kernel functions, such as printk, kmalloc, kfree, but only the functions that are EXPORTed by the kernel(using EXPORT(symbol_name))
- Modules (or any kernel code for that matter) cannot call user-space library functions, such as malloc, free, printf etc.
- Modules should not include standard header files, such as stdio.h, stdlib.h, etc.
- Segmentation fault may be harmless in user space, but a kernel fault can crash the entire system
- Version Dependency: Module should be recompiled for each version of kernel that it is linked to

7.2.5 Concurrency Issues

- Many processes could try to access your module concurrently. So different parts of your module may be active at the same time
- Device interrupts can trigger Interrupt Service Routines (ISR), ISRs may access common data that your module uses as well

- Kernel timers can concurrently execute with your module and access common data
- You may have symmetric multi-processor (SMP) system, so multiple processors may be executing your module code simultaneously (not just concurrently).
- Therefore, your module code (and most kernel code, in general) should be re-enterant, Capable of correctly executing correctly in more than one context simultaneously

7.2.6 Error Handling

```
int __init my_init_function(void)
{
    int err;

    /* registration takes a pointer and a name */
    err = register_this(ptr1, "skull");
    if (err) goto fail_this;
    err = register_that(ptr2, "skull");
    if (err) goto fail_that;
    err = register_those(ptr3, "skull");
    if (err) goto fail_those;

    return 0; /* success */

 fail_those: unregister_that(ptr2, "skull");
 fail_that: unregister_this(ptr1, "skull");
 fail_this: return err; /* propagate the error */
}
```

```
void __exit my_cleanup_function(void)
{
    unregister_those(ptr3, "skull");
    unregister_that(ptr2, "skull");
    unregister_this(ptr1, "skull");
    return;
}
```

- In case of failure, undo every registration activity
- But only those that were registered successfully

7.2.7 Module Parameters

- Command line: insmod hellon.ko howmany=10 whom="Class"
- Module code has:

```
static char *whom = "world";
static int howmany = 1;

module_param(howmany, int, S_IRUGO);
```

```
module_param(whom, charp, S_IRUGO);
```

7.3 Character devices in Linux

7.3.1 Device Classification

- Character (char) devices
  - byte-stream abstraction(keyboard, mouse)
- Block devices
  - reads/writes in fixed block granularity (hard disks, CD drives)
- Network devices: stream of packets
  - message abstraction, send/receive packets of varying sizes (network interface cards)
- Others
  - USB, SCSI, Firewire, I2O
  - Can (mostly) be used to implement one or more of the above three classes

7.3.2 "Miscellaneous" Devices in Linux

- These are character devices used for simple device drivers.
- All miscellaneous devices share a major number (10).
- But each device gets its own minor number, requested at registration time

7.3.3 Implementing a device driver for a miscellaneous device

1. Declare a device struct

```
    static struct miscdevice my_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "my device",
    .fops = &my_fops
};
```

2. Declare the file operations struct

```
    static struct file_operations my_fops = {
        .owner = THIS_MODULE,
        .open = my_open,
        .release = my_close,
        .read = my_read,
        ...
        .llseek = noop_llseek
    };
    // The function pointers that are not initialized
        above will be assigned some sensible default value
        by the kernel.
```

3. register the device with kernel, usually in the module initialization code

```
    static int __init my_module_init()
    {
        ...
        misc_register(&my_misc_device);
        ...
    }
    // And don't forget to unregister the device when
        removing the module
    static void __exit my_exit(void)
    {
        misc_deregister(&my_misc_device);
        ...
    }
```

4. Implement the fops functions

```
static ssize_t my_read(struct file *file, char __user
    * out, size_t size, loff_t * off)
{
    ...
    sprintf(buf, "Hello World\n");
    copy_to_user(out, buf, strlen(buf)+1);
    ...
}
```

5. Warning
    - allocate memory for buf
    - Check if "out" points to a valid user memory location using access_OK()
    - check for errors during copy_to_user()

7.3.4 How do file ops work on character devices

- A file operation on a device file will be handled by the kernel module associated with the device
- Use "open()" system call to open "mydevice" file
    - fd = open("/dev/mydevice", O_RDWR);
    - opens /dev/mydevice device for read and write operation.
    - OS will call my_open() file operation handler in the kernel module which is associated with the device.
    - misc_register(&my_misc_device) in my_module_init() registers the character device. It creates an entry in the "/dev" directory for "mydevice" file and informs the operating system what file-operations handler functions are available for this device.
- Use "read()" system call to read from the "mydevice" file
    - n = read(fd, buffer, size);
    - finally calls the my_read() function passed through the fops structure in your kernel module

7.3.5 Moving data in and out of the Kernel

- copy_to_user()
  - unsigned long copy_to_user (void __user * dst, const void * src, unsigned long n)
  - Copies data from kernel space to user space
  - Returns number of bytes that could not be copied. On success, this will be zero.
  - Checks that dst is writable by calling access_ok on dst with a type of VERIFY_WRITE. If it returns non-zero, copy_to_user proceeds to copy
- copy_from_user()
  - unsigned long copy_from_user (void * dst, const void __user * src, unsigned long n)
  - Copies data from user space to kernel
  - Returns number of bytes that could not be copied. On success, this will be zero

7.3.6 Memory allocation/deallocation in Kernel

- Memory Allocation:
  - kmalloc(): Allocates physically contiguous memory:

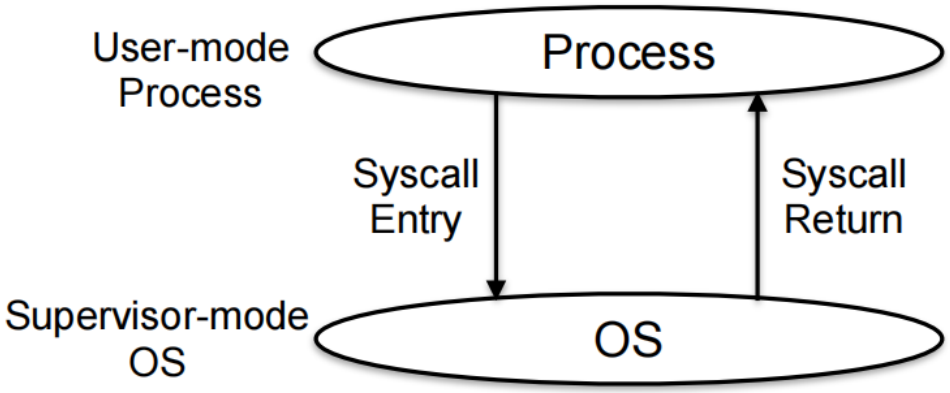
```
void * kmalloc(size_t size, int flags)
```

  - kzalloc(): Allocates memory and sets it to zero
  - vmalloc(): Allocates memory that is virtually contiguous and not necessarily physically contiguous.
- Memory Deallocation: kfree()

8 System Calls

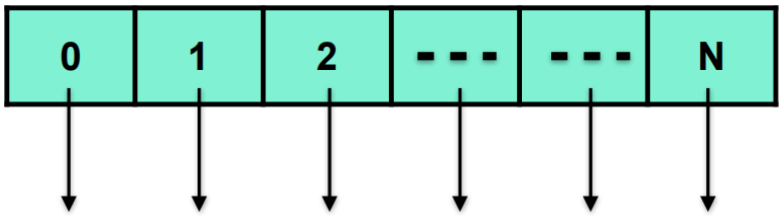
8.1 Definition

- Interface to allow User-level processes to safely invoke OS routines for privileged operations
- Safely transfer control from lower privilege level (user mode) to higher privilege level (supervisor mode), and back



8.1.1 System Call table

- Protected entry points into the kernel for each system call: We don't want application to randomly jump into any part of the OS code
- Syscall table is usually implemented as an array of function pointers, where each function implements one system call
- Syscall table is indexed via system call number



8.1.2 System Call Invocation

1. System calls is invoked via a special CPU instruction: The system call number and arguments passed via CPU registers and optionally stack
2. CPU saves process execution state
3. CPU switches to higher privilege level: jumps to an entry point in OS code
4. OS indexes the system call table using the system call number
5. OS invokes the system call via a function pointer in the system call table
    - For performance reasons, the system call usually executes in the execution context of the calling process, but in privileged mode
    - Some OS may execute the system call in a separate execution context for better security
6. If the syscall involves blokcing I/O, the calling process may block while the I/O completes
7. When syscall completes, the calling process is moved to ready state
8. The saved process state is restored
9. Processor switches back to lower privilege level using SYSEXIT/iret instructions
10. Process returns from the system call and continues.

User process	Invoke syscall using, say, SYSENTER instruction (arguments in registers/stack)
CPU	Switch CPU to supervisor mode. Jump to entry point in kernel.
Kernel	Save process state Lookup Syscall table. Invoke syscall.
Kernel	Optionally Block process if it needs to wait for I/O or other events. Return process to ready state when woken.
Kernel	Restore saved process state SYSEXIT
CPU	Switch CPU to user mode Return to user process
User Process	Return from system call. Continue

8.2 Syscall Usage

- To make it easier to invoke system calls, OS writers normally provide a library that sits between programs and system call interface: Libc, glibc, etc
- This library provides wrapper routines
- Wrappers hide the low-level details of
  - Preparing arguments
  - Passing arguments to kernel
  - Switching to supervisor mode
  - Fetching and returning results to application
- Helps to reduce OS dependency and increase portability of programs

8.3 Implementing

8.3.1 Steps in Writing a System Call

1. Create an entry for the system call in the kernel's syscall_table: User processes trapping to the kernel (through SYS_ENTER or int 0x80) find the syscall function by indexing into this table

2. Write the system call code as a kernel function

   - Be careful when reading/writing to user-space
   - Use copy_to_user() or copy_from_user() routines. These perform sanity checks.

3. Implement a user-level wrapper to invoke your system call: Hides the complexity of making a system call from user applications.

8.3.2 Code

1. Create a sys_call_table entry (for 64-bit x86 machines):Syscall table initialized in arch/x86/entry/syscall_64.c

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
...
309 common getcpu sys_getcpu
310 64 process_vm_readv sys_process_vm_readv
311 64 process_vm_writev sys_process_vm_writev
312 common kcmp sys_kcmp
313 common foo sys_foo
```

2. Write the system call handler

   - System call with no arguments and integer return value

```
SYSCALL_DEFINE0(foo){
    printk (KERN ALERT "sys_foo: pid is %d\n",
        current->pid);
    return current->pid;
}
```

- Syscall with one primitive argument

```
SYSCALL_DEFINE1(foo, int, arg){
    printk (KERN ALERT "sys_foo: Argument is
        %d\n", arg);
    return arg;
}
```

- To see system log:
  - dmesg
  - less /var/log/kern.log

Verifying argument passed by user space

```
SYSCALL_DEFINE1(close, unsigned int, fd)
{
        struct file * filp;
        struct files_struct *files = current->files;
        struct fdtable *fdt;
        spin_lock(&files->file_lock);
        fdt = files_fdtable(files);

        if (fd >= fdt->max_fds)
                goto out_unlock;
        filp = fdt->fd[fd];
        if (!filp)
                goto out_unlock;
        …
out_unlock:
        spin_unlock(&files->file_lock);
        return -EBADF;
}
```

- Call-by-reference argument
  - User-space pointer sent as argument.
  - Data to be copied back using the pointer.

```
SYSCALL_DEFINE3( read, unsigned int, fd,
        char __user *, buf,  size_t, count)
{
        …
    if( !access_ok( VERIFY_WRITE, buf, count))
        return –EFAULT;
        …
}
```

3. Invoke syscall handler from user space
   - Use the syscall(...) library function.
   - For instance, for a no-argument system call named foo(), you'll call

```
ret = syscall(__NR_sys_foo);
```

- For a 1 argument system call named foo(arg), you call

```
ret = syscall(__NR_sys_foo, arg);
```

- and so on for 2, 3, 4 arguments etc.

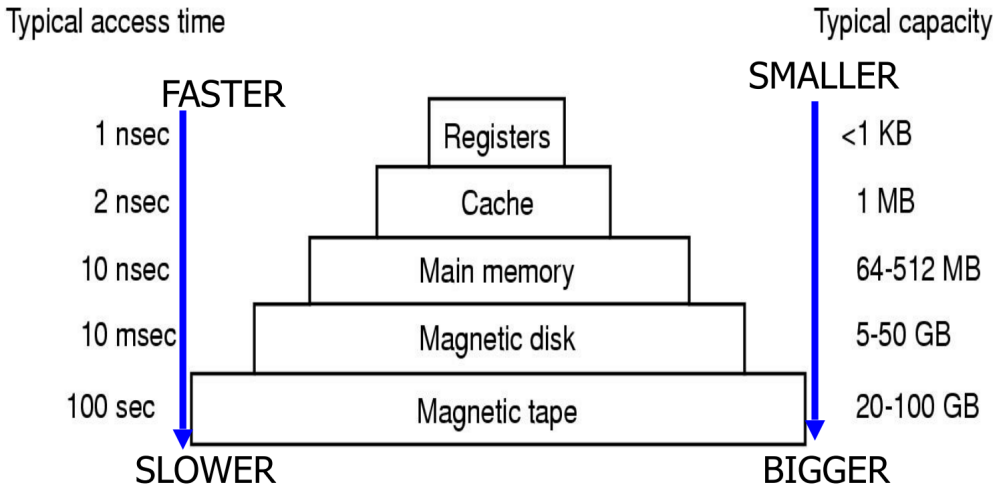
9 Memory Management

Ideally programmers want memory that is

- Large
- Fast
- Persistent

9.1 Memory Hierarchy

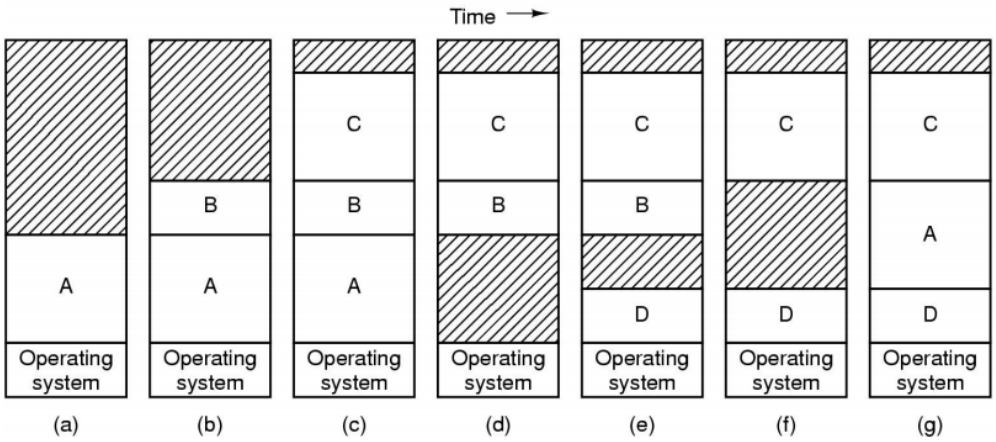
- Registers & Cache: small amount of fast, expensive, volatile memory
- Main memory: some medium-speed, medium price, volatile/persistent memory, DRAM
- Disk & Tape: Lots of slow, cheap, persistent, storage



9.2 Relocation and Protection

- Problem: A programmer doesn't know where a program will be loaded in memory
  - address locations of variables and code routines cannot be absolute
  - must keep a program out of other processes' partitions
- Solution: Use base and limit values
- Relocation
  - Address locations in a program are relative
  - They are added to a base value to map to physical addresses
- Protection: Access to address locations larger than limit value results in an error

9.3 Swapping



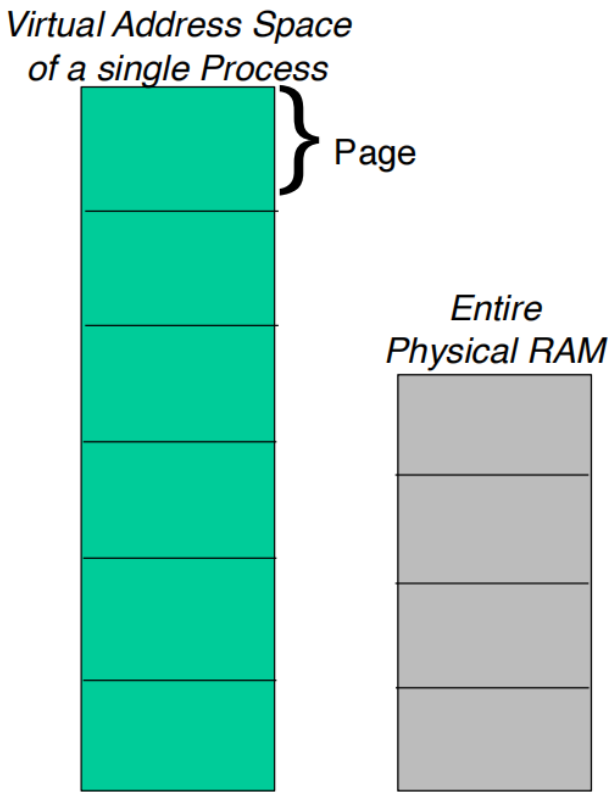
- Physical memory may not be enough to accommodate the needs of all processes
- Memory allocation changes as
  - processes come into memory
  - leave memory and are swapped out to disk

  - Re-enter memory by getting swapped-in from disk

- Shaded regions are unused memory

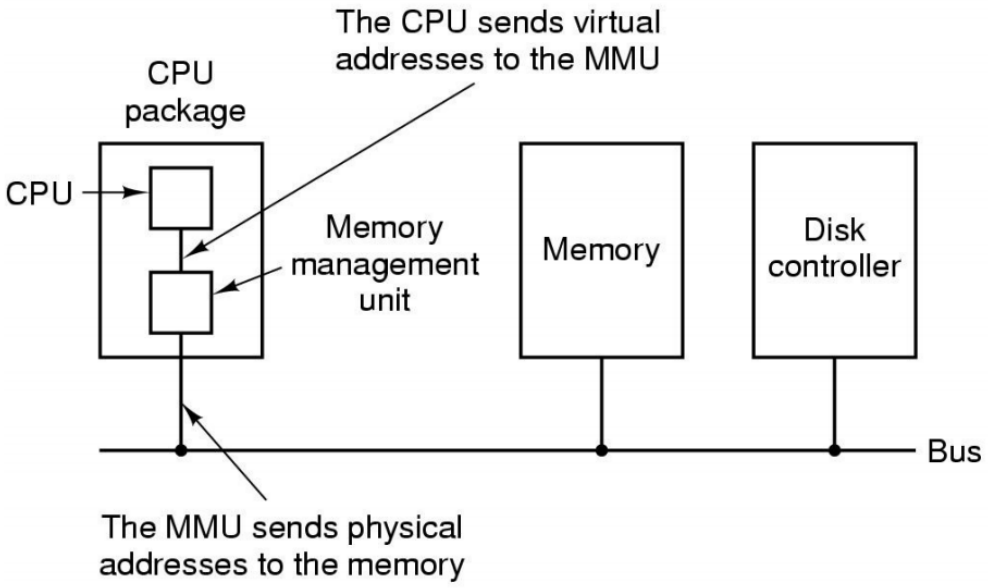
9.4 Paging

- Swapping the memory of an entire process is useful when the sum of memory needed by all processes is greater than the total RAM available in the system.
- But sometimes, a single process might require more memory than the total RAM in the system.
- In such cases swapping an entire process is not enough.
- Rather, we need to break up the memory space of a process into smaller equal-sized pieces, called PAGES.
- OS then decides which pages stay in memory and which get moved to disk.
- Virtual memory: means that each process gets an illusion that it has more memory than the physical RAM in the system



9.5 Memory Management Unit (MMU)

- MMU is a hardware module that accompanies the CPU
- It translates the Virtual Address used by executing instructions to Physical Addresses in the main memory
- It need to be very quick so it stands along with CPU
- Steps
  1. CPU tells MMU the virtual address P it needs
  2. MMU translante P into P* which is the physical address
  3. P* is sent through Bus to the memory controller
  4. memorry controller get the P* data and send it back by Bus

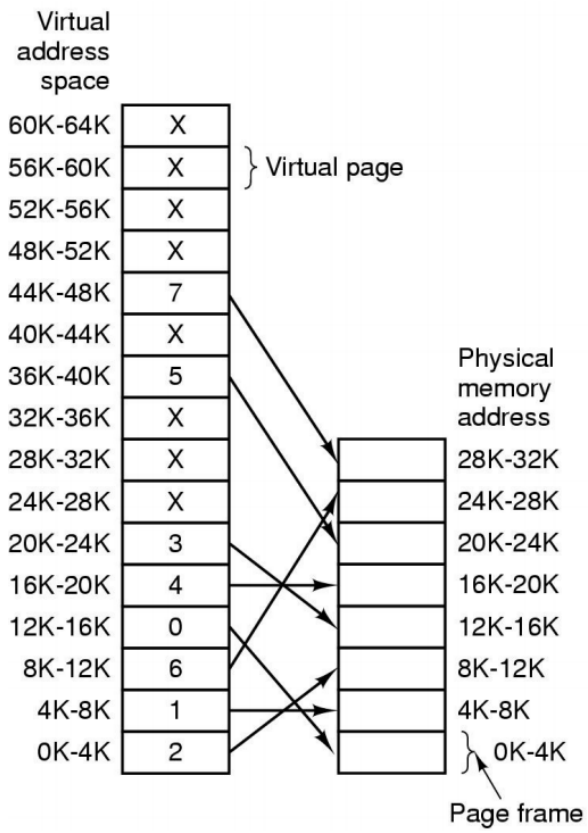


9.6 Size of address space (in bytes) as a function of address size (in bits)

Number of bits in address	Maximum address space size (bytes)
0	2^0 = 1 byte
1	2^1 = 2 bytes
2	2^2 = 4 bytes
10	2^{10} = 1024 = 1KiB
12	2^{12} = 4KiB
16	2^{16} = 64 KiB
32	2^{32} = 4GiB (Gibibytes)
64	2^{64} = 16 EiB (Exbibytes)

9.7 PageTable

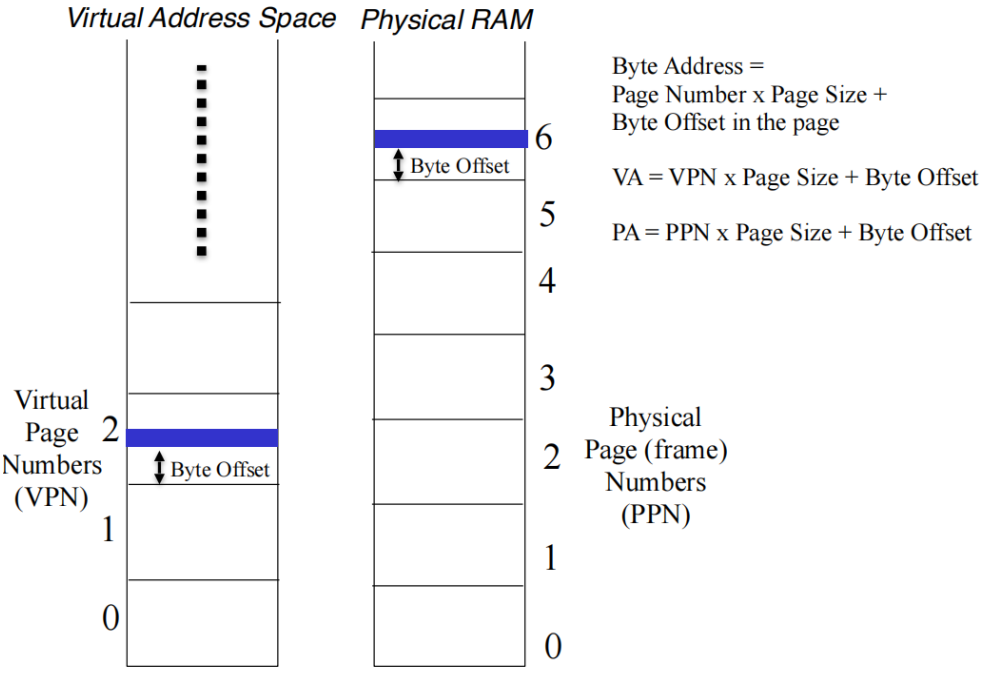
- An array that stores the mapping from virtual page numbers to physical numbers
- The OS maintains
  - One page table per userspace process
  - And usually another page table for kernel memory



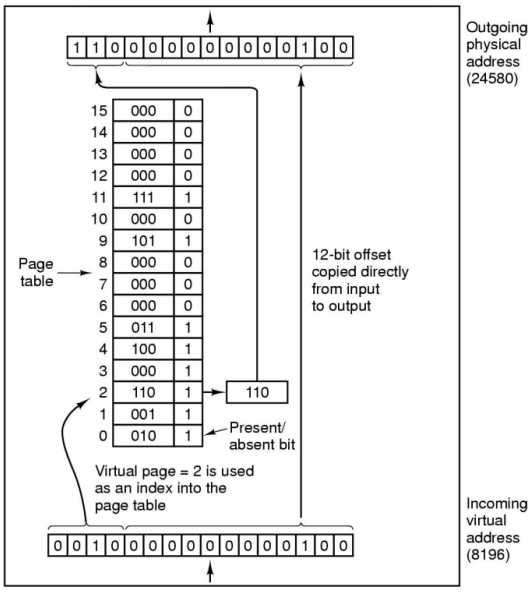
9.8 Traslate Virtual address (VA) to physical address (PA)

- Byte Address = Page Number x Page Size + Byte Offset in the page
- VA = VPN x Page Size + Byte Offset

- PA = PPN x Page Size + Byte Offset



9.8.1 Virtual Address Translation For Small Address Space

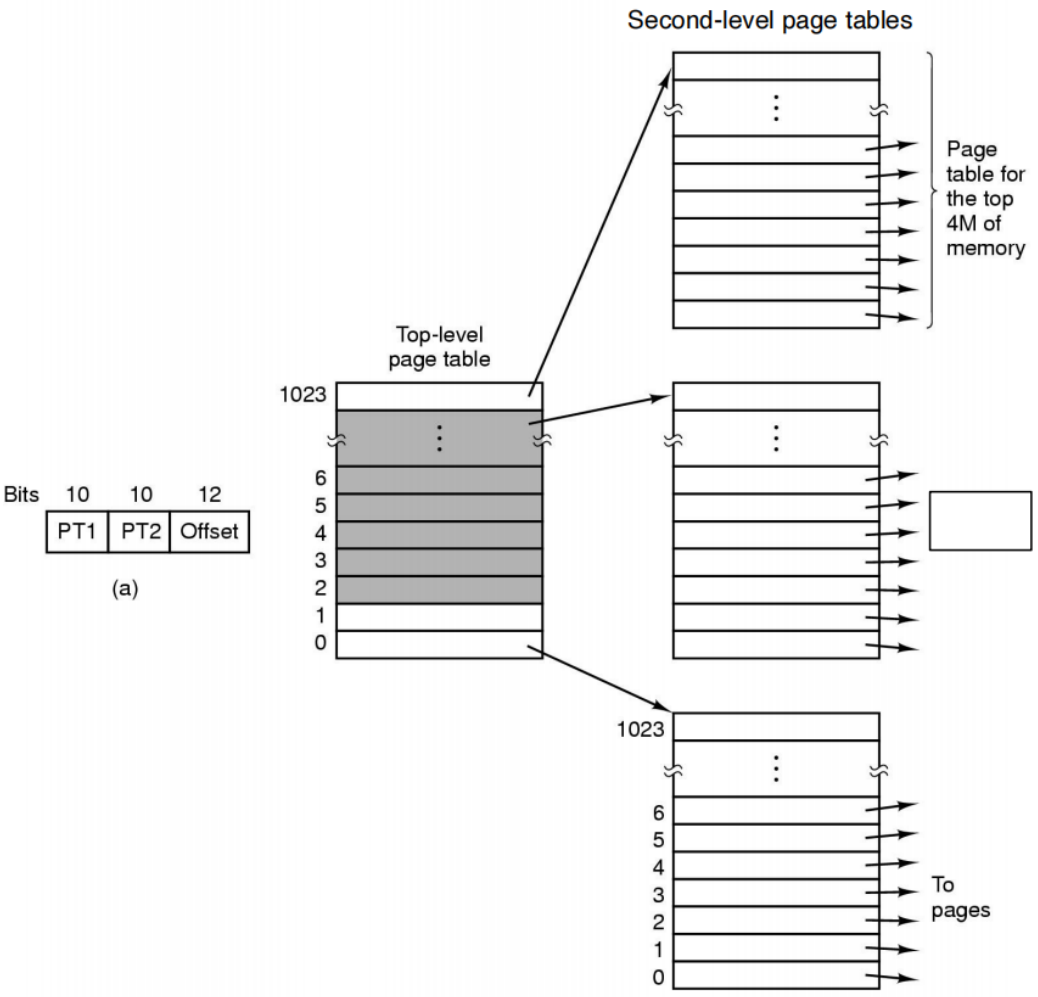


Internal operation of MMU with 16 4 KB pages

- The first 12 bits: byte Offset
- The last 4 bits: virtual page number
- Page fault: the absent bit in page table is 0
  - OS would fix the entry
  - Rerun the instruction
- Each process has its own page table, it used to need to be load in the MMU, but as the bit of computer grows, the MMU need to be got out to the DRAM.
  - page table's entry number becomes larger and larger: a 32-bit computer has over 1 million entries for each process's page table, which significantly decreases the speed of switching context.
  - the storage cost is too expensive

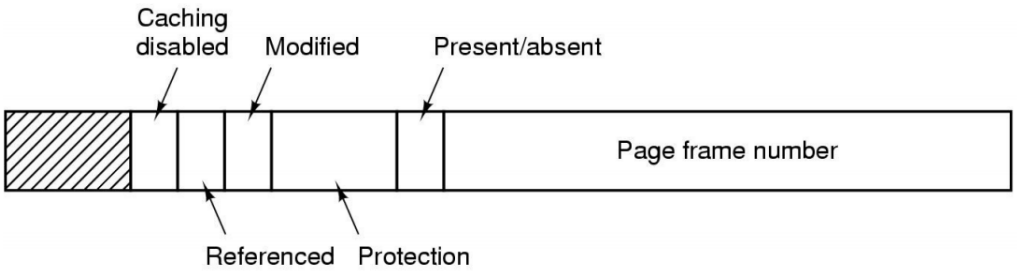
- so MMU keeps a pointer(mostly a register) to the DRAM which is the base of physical memory address
- Problem: what if the contiguous memory is hard to find? Different layer of page table
- Problem: what if the translation is inefficient? Store a small page table in MMU in the cache, it's called TLB. If we can find the physical memory in TLB, it's a hit. Otherwise it's a miss, go to DRAM.

9.8.2 Virtual Address Translation For Large Address Space



- 32 bit address with 2 page table fields
- Two-level page tables
- PT too Big for MMU: Keep it in main memory
- But how does MMU know where to find PT? Registers (CR2 on Intel)

9.9 Typical Page Table Entry (PTE)



- Page Frame number = physical page number for the virtual page represented by the PTE
- Referenced bit: Whether the page was accessed since last time the bit was reset.
- Modified bit: Also called "Dirty" bit. Whether the page was written to, since the last time the bit was reset.
- Protection bits: Whether the page is readable? writeable? executable? contains higher privilege code/data?
- Present/Absent bit: Whether the PTE contains a valid page frame number. Used for marking swapped/unallocated pages
- Cashing disabled bit: point to some I/O devices(memory map I/O) rather than memory

9.10 TLBs – Translation Lookaside Buffers

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

- TLB is a small cache that speeds up the translation of virtual addresses to physical addresses.
- TLB is part of the MMU hardware (comes with CPU)
- TLB is made of fully associative cache, which could compare all the entries at the same time.
- It is not a Data Cache or Instruction Cache. Those are separate.
- TLB simply caches translations from virtual page number to physical page number so that the MMU don't have to access page-table in memory too often.
- On older x86 processors, TLB had to be "flushed" upon every context switch because there is no field in TLB to identify the process context. Tagged TLB can reduce this overhead, it use ASID to remember its owner process.
- Spatial Locality: if the process is accessing i byte, it's very likely it will access nearby bytes.
- Temporal Locality: if the process is accessing i byte, it's very likely it will keep accessing i byte in the future
- If CPU do a context switch, all the valid bits will be set to 0, and the new process needs to take some time to get in a new page table. There will be misses at first, but as PLB warms up, it will be hit eventually.

9.11 Impact of Page Size on Page tables

9.11.1 Small Page Size

- Advantages
  - less internal fragmentation(space wasted)
  - page-in/page-out less expensive
- Disadvantages
  - process that needs more pages has larger page table
  - Smaller "TLB Coverage" (next slide)

10 TLB and TLB Coverage

- Working Set: The amount of memory that my program actually needs.
- TLB Coverage: The amount of memory that my TLB has.
- Goal: TLB Coverage > Working Set
  - increase page size
  - increase entry number

10.1 Cold Start Penalty

- Cost of repopulating the TLB (and other caches) upon a context switch.
- Immediately after a context switch, all (or many) of TLB entries are invalidated.
  - On some x86 processors, TLB has to be "flushed" upon every context switch because there is no field in TLB to identify the process context.
- Every memory access by the newly scheduled process may results in a TLB miss
- MMU must then walk the page-table in main memory to repopulate the missing TLB entry, which takes longer than a cache hit.

10.2 TLB Coverage

- Max amount of memory mapped by TLB: Max mount of memory that can be accessed without TLB misses
- TLB Coverage = N x P bytes
  - N = Number of entries in TLB
  - P = Page size in bytes
  - N is fixed by hardware constraints
  - So, to increase TLB Coverage, we must increase P

10.3 Tagged TLB

- A "tag" in each TLB entry identifies the process/thread context to which the TLB entry belongs
- Thus TLB entries for more than one execution context can be stored simultaneously in the TLB.
- TLB lookup hardware matches the tag in addition to the virtual page number.
- With tags, context switch no longer requires a complete TLB flush.
- Reduces cold-start penalty.

10.4 Two types of memory translation architectures

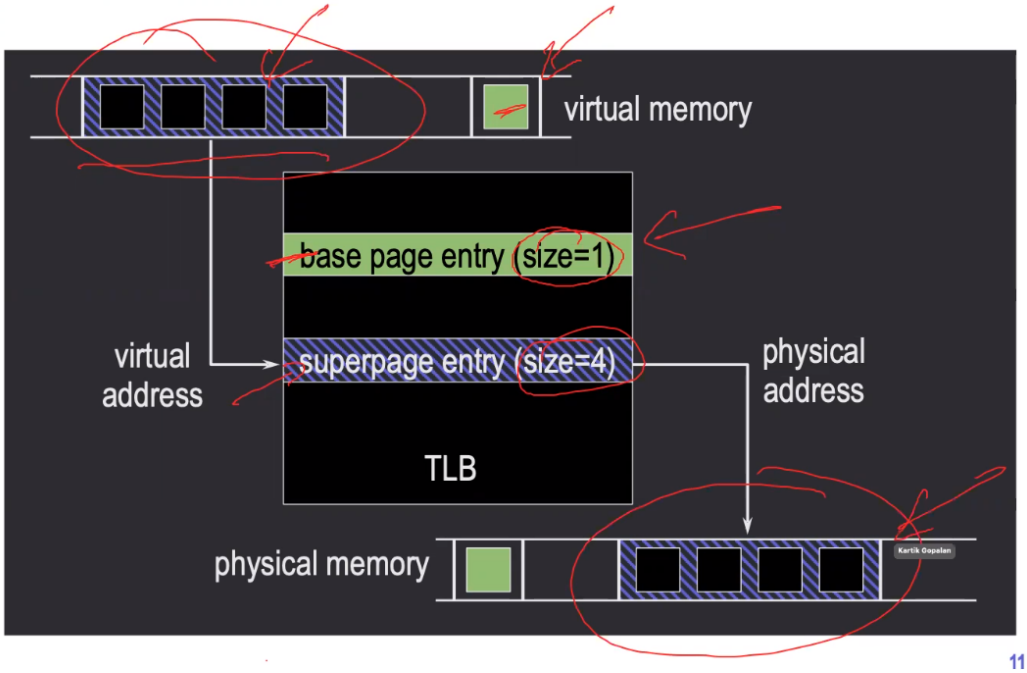
- Architected Page Tables: hardware decides how the page table looks like and determines the way MMU do page table walk
  - Page table interface defined by ISA and understood by memory translation hardware
  - E.g. x86 architecture
  - MMU handles TLB miss (in hardware)
  - OS handles page faults (in software)
  - ISA specifies page table format

- Architected TLBs: throw an exception when miss rather than do page table walk
  - TLB interface defined by ISA and understood by MMU
  - E.g. alpha architecture
  - TLB miss handled by OS (in software)
  - ISA does not specify page table format

10.5 Superpages/Hugepages/Largepages

10.5.1 Superpages

- Memory pages of larger sizes than standard pages: supported by most modern CPUs
- Superpage size = power of 2 x the base page size
- Only one TLB entry per superpage. But multiple (identical) page-table entries, one per base page
- Constraints:
  - contiguous (physically and virtually), otherwise the baseless offset mechanism won't work
  - aligned (physically and virtually) the super page size: avoid external fragmentation
  - uniform protection attributes: only one entry in the TLB only has one protection bit
  - one reference bit, one dirty bit



- page table entry
  - normal page: size 1
  - superpage: size n
- page table virtual page number
  - normal page: v1
  - superpage: all the entries have the same vitrual page number, all the entries are identical.
- The hundred superpage size four = the first page is the 400 page in the physical memory.

10.5.2 Restrictions Examples

- Size restrictions
  - 8KiB = 21 x 4KiB
  - 16KiB = 22 x 4KiB

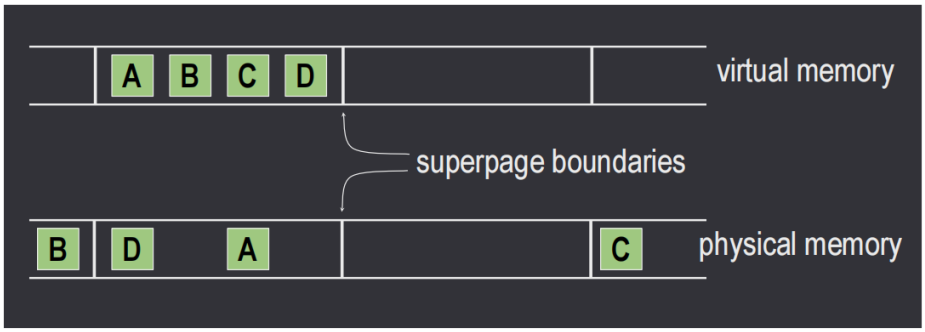
- 32KiB = 23 x 4KiB etc

- Contiguity restrictions for superpage of size 4
  - Possible: Base pages 8,9,10,11 in one superpage
  - Impossible: Base pages 8, 10, 20, 22 in one superpage

- Alignment restrictions for superpage of size 4
  - Possible
    - Base pages 4,5,6,7 in one superpage
    - Base pages 8,9,10,11 in one superpage
    - Base pages 12,13,14,15 in one superpage
  - Impossible
    - Base pages 6,7,8,9 in one superpage
    - Base pages 13,14,15,16 in one superpage

10.5.3 Problems

1. superpage allocation: How / when / what size to allocate?



   - allocate process anywhere and bring them together later, will waste CPU cycles
   - reserve some space for some pages assuming that they would be used as a superpage.

2. promotion: create a superpage out of a set of smaller pages. When to promote?

   - wait for the application to touch these pages, avoid promoting the size 4 superpage to size 8, but may lose the opportunity to increase TLB Coverage.
   - create a small superpage, may incur some overhead
   - forcibally populate pages, may cause I/O cost or increase internal fragmentation

3. demotion: convert a superpage into smaller pages: when page attributes of base pages of a superpage become non-uniform?

4. fragmentation

   - External fragmentation
     - use of multiple page sizes
     - Scattered wired pages
   - Contiguity of free pages is a contended resource
     - Allocating a superpage requires that sufficient number of contiguous base pages must be free.
   - OS must
     - use contiguity restoration techniques
     - trade off impact of contiguity restoration against superpage benefits

10.5.4 Design

1. Key observation: Once an application touches the first page of a memory object then it is likely that it will quickly touch every page of that object

- Example: array initialization
- Opportunistic policies
  - superpages as large and as soon as possible
  - as long as no penalty if wrong decision
- Superpage allocation: Preemptible reservations: Assume the process will use a superpage and reserve the space for a short time, if it doesn't, give these space to another process.

2. Allocation: reservation size

   - Opportunistic policy
     - Go for biggest size that is no larger than the memory object (e.g., file)
     - If required size not available, try preemption before resigning to a smaller size

3. Incremental promotions: Promotion policy: opportunistic

4. Speculative demotions

   - One reference bit per superpage
   - On memory pressure, demote superpages when resetting ref bit
   - Re-promote (incrementally) as pages are referenced
   - Demote also when the page daemon selects a base page as a victim page
   - Dirty superpages
     - One dirty bit per superpage
     - Demote on first write to clean superpage
     - Re-promote (incrementally) as other pages are dirtied

5. Fragmentation control

   - Low contiguity: modified page daemon for victim selection
     - restore contiguity: move clean, inactive pages to the free list
     - minimize impact: prefer victim pages that contribute the most to contiguity

- Cluster wired pages
  - Assign a dedicated region of physical memory for wired pages
  - So that they break contiguity for superpage allocations from rest of the memory

11 The UNIX Time-Sharing System

11.1 UNIX overview

- Unix is a general-purpose, multi-user, interactive operating system
- Originally developed for DEC PDP-7, -9, and -11 computers
- Written in C language
- Now widely supported across almost all hardware platforms in various variants: System V, FreeBSD, Linux, Solaris etc

11.1.1 Major Innovations

- Hierarchical file system
- Compatible file, device, and inter-process I/O
- Background (asynchronous) and foreground processes
- Interactive Shell

11.2 File System

Most important role of UNIX is to provide a file-system. There are three types of files

- Ordinary files: No particular structure imposed by OS
- Directories: Mapping between filenames and files
- Special files: I/O devices

11.2.1 Hard Links

link to the real file.

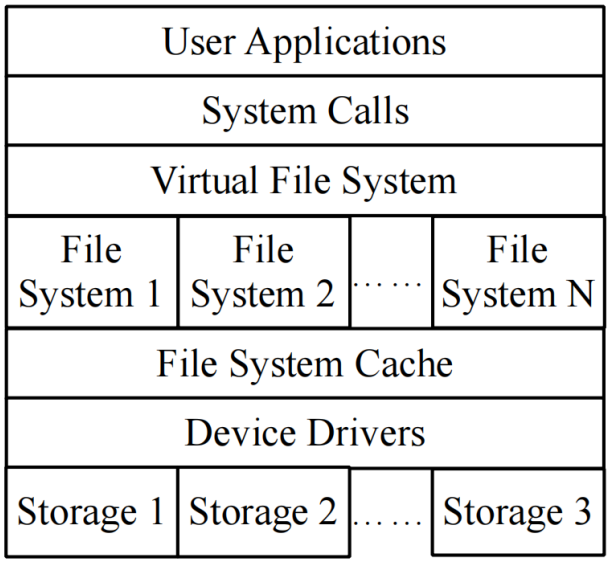
11.2.2 What is a File System?

- File system is the OS component that organizes data on the raw storage device.
- Data, by itself, is just a meaningless sequence of bits and bytes.
- Metadata is the information that describes the data.
  - Also called attributes.
  - Without meta-data, data itself will be incomprehensible.
- Responsibilities of a file system
  - Defines the format of the data objects.
  - Defines the format and meaning of meta-data associated with each data object. E.g. File name, permissions, and size of a file.
  - Manages the location of the individual data blocks of each data object.
  - Manages free space on the storage device.

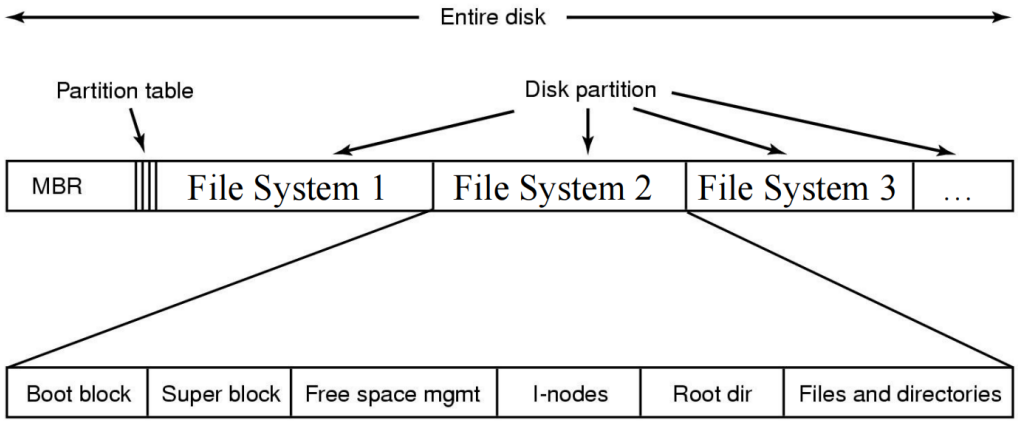
11.2.3 Virtual File System (VFS)

VFS provides

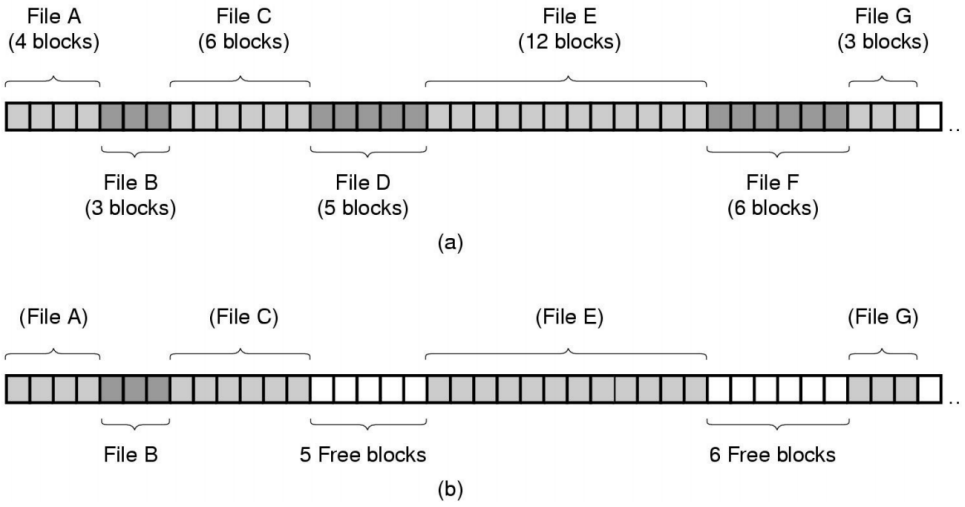
- A common system call interface to user applications to access different file systems implemented in the OS.
- A common interface to file systems to "plug into" the operating system and provide services to user applications.



11.2.4 Partitions and File-system Layout



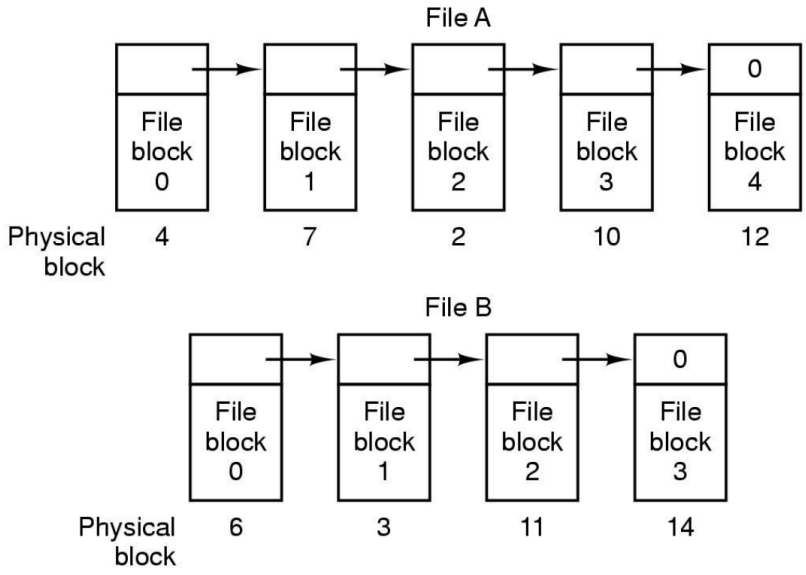
11.2.5 Organizing Files on Disk: Contiguous Allocation



(a) Contiguous allocation of disk space for 7 files

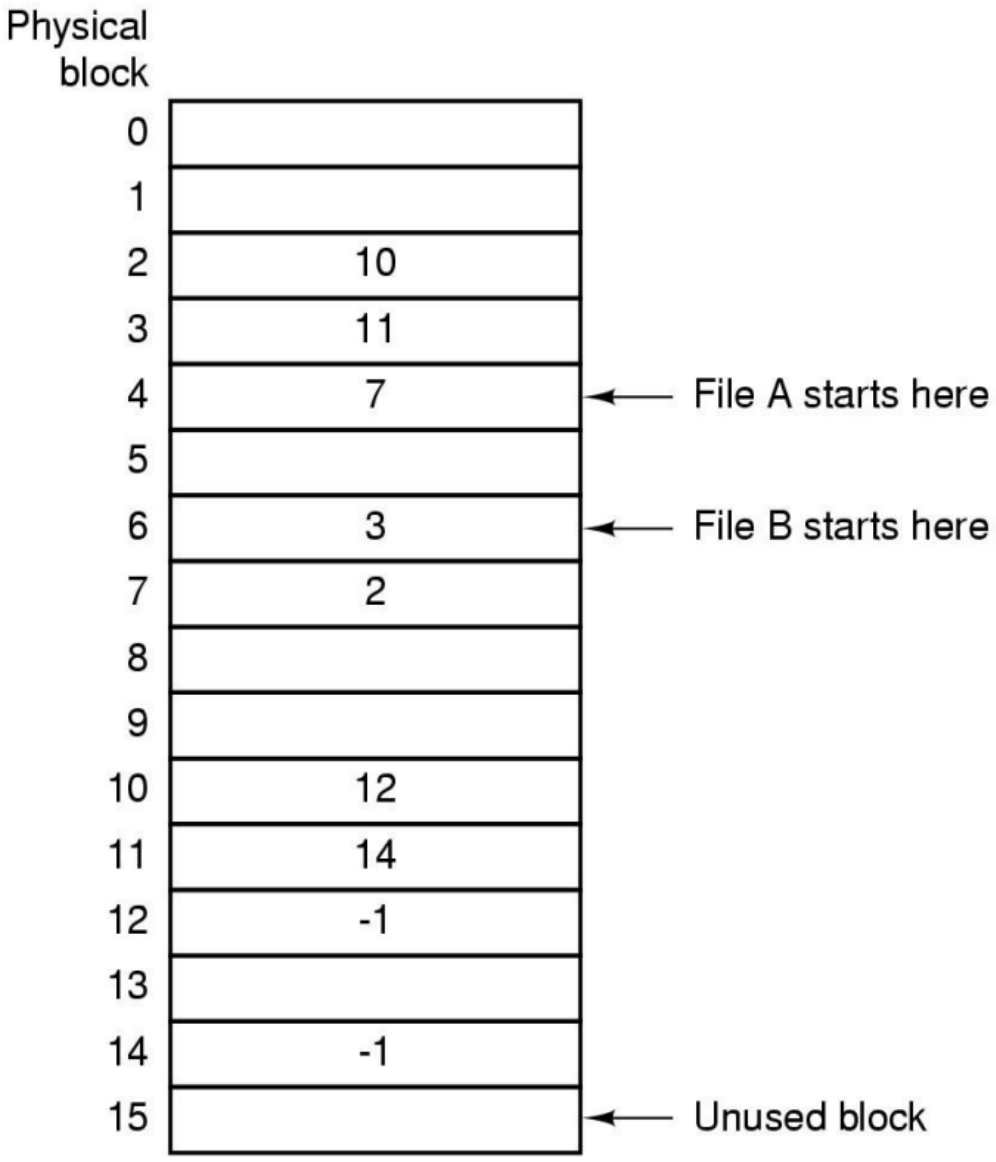
(b) State of the disk after files D and E have been removed

11.2.6 Organizing Files on Disk: Singly Linked List of Blocks



- Advantage: Logically contiguous blocks can be discontiguous on disk
- Disadvantage: Random seeks are expensive. Requires traversal from the start.

11.2.7 Organizing Files on Disk: File Allocation Table

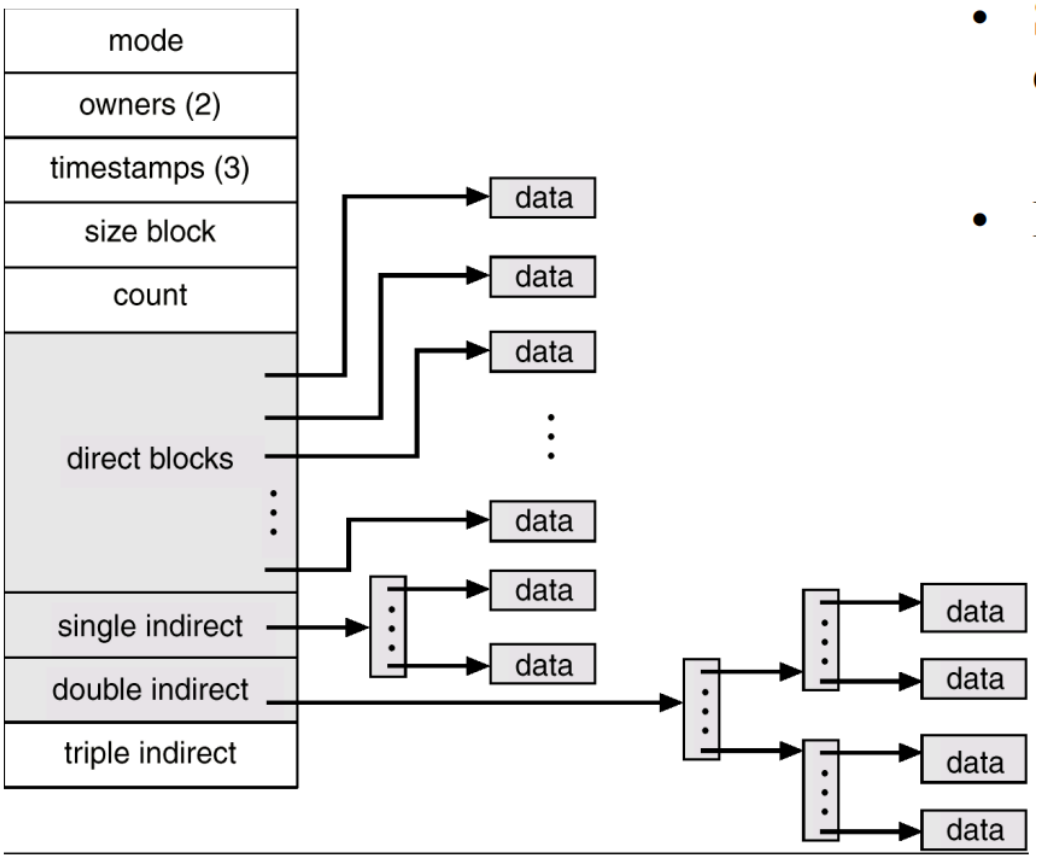


- Linked list allocation using a file allocation table in RAM
- Doesn't need a separate "next" pointer within each block
- But random seeks are still expensive

11.2.8 i-nodes (index nodes)

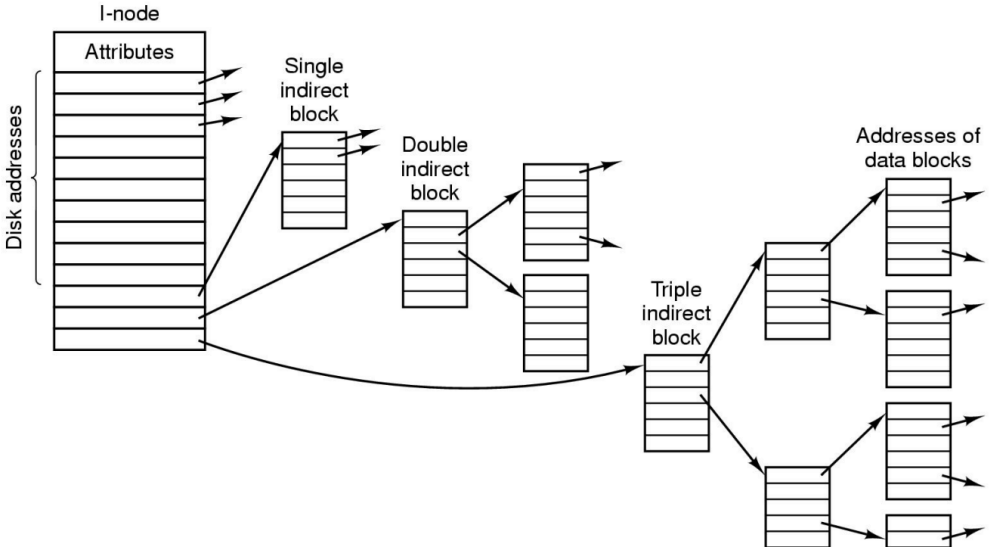
- Each file is described by an i-node
- i-node stores the metadata for the file
- Metadata = file info + location of data blocks
- i-nodes are stored on the disk
- An i-node is to a file what a page-table is to a virtual address space
  - Page table maps: virtual page number in a virtual address space → physical page frame number in DRAM
  - i-node maps: logical block number in a file → physical block location on disk

11.2.9 Unix i-node (index node)



- Small files can be accessed quickly
- If each block is 4KB
  - First 48KB of file reachable from 12 direct blocks
  - Next 4MB available from single-indirect blocks
  - Next 4GB available from double-indirect blocks
  - Next 4TB available through the triple-indirect blocks
- Any block can be found with at most 3 disk accesses

11.2.10 Another view of a UNIX i-node



- The whole data structure above is ONE i-node for ONE file
- i-node grows dynamically as the file grows
- Just like page-tables, i-node tracks a giant array broken up into many pieces

11.2.11 Special files

- "most unusual feature"
- Located in /dev Just like ordinary files.
- But read/write requests result in activation of the associated device.
- Advantages of treating I/O devices as files
  - File and device I/O are similar
  - File and device names have same syntax and meaning.
  - Programs that operate on files can also operate on devices.
  - Same protection mechanisms as regular files

11.2.12 Removable file system

- Different parts of filesystem can be on different devices
- mount: Allows a storage device to be referenced as a subtree of existing rooted file system.
- Hard links across mounted file systems disallowed. Because hard links happen on the level of i-node, and different file systems have different ways of representing i-node.

11.2.13 Protection

- Each user has a userid
- Files marked as owned with userid of the creator
- Seven protection bits
  - Six bits for read, write, and execute permissions for user, group, and others
  - 7th bit → set-user-ID bit
    - Temporarily change the userid of current user to the owner when file is executed as a program.
    - Allows the safe use of privileged programs that require access to special system files (e.g. system logs).
    - Actual userid of the invoker is available to the program for credential checks.

11.2.14 I/O calls

- filep = open (name, flag): Create system call creates and opens a new file. Truncates to zero if file exists. filep is a file descriptor. Notion of a file descriptor hasn't changed over the years
- No locking provided by OS for multi-user access
  - Lets users figure out synchronization.
  - Locks are "neither necessary nor sufficient". Why?

  - Internal OS locks for consistency of data structures

- n = read(filep, buffer, count): Read returns 0 when end of file (EOF) reached
- n = write(filep, buffer, count): Write beyond end of file grows the file automatically
- location = seek(filep, base, offset): For random I/O

11.2.15 Processes and images

- processid = fork(label)
- filep = pipe( )
- execute(file, args, argo, ..., arg,+)
- processid = wait( )
- exit (status)

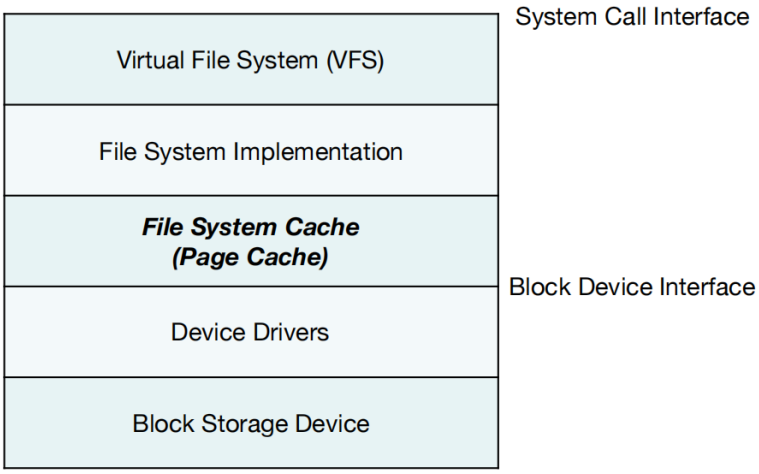
11.2.16 Shell

- Triggered upon login by the init process
- Can be replaced by other commands: command arg1 arg2 • • - argn
- ls > there
- ed > script
- Filters: ls | grep foo | wc -l
- Multitasking
  - ls; ed
  - as source > output &
  - as source > output & ls > files &
  - (date; ls) > x &

11.2.17 read/write buffering

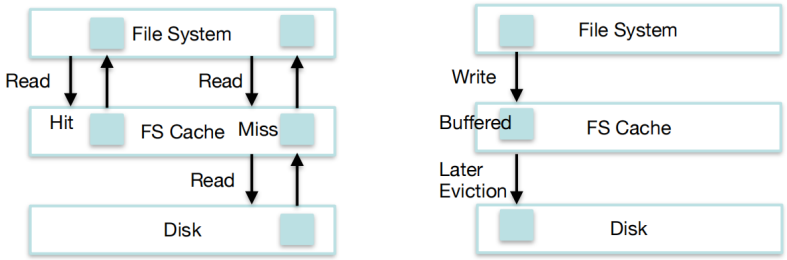
- System buffers to hide block I/O activity
- Users can read/write at any byte granularity
- OS converts these to block-level I/O
- Disadvantage: if the power fails, all the data on the memory would be lost.
- Advantage: save IO resource

11.3 File System Cache



FS Cache is a part of main memory used to store frequently accessed data blocks from the disk

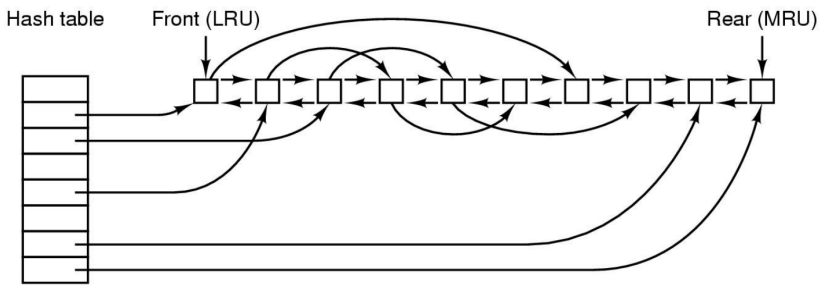
11.3.1 How it works



- Before accessing the disk, look in the FS cache.
- If data block is in FS cache, no need to go to disk.
- Periodically, purge the cache of infrequently used data blocks.
- Readahead: If application asks for block i, cache would automatically read i+1, i+2, i+3... block.
- Disk also do readahead to its disk cache

Claim: If the cache works well, then most I/O accesses to the physical disk will be writes. Why?

11.3.2 Data Structure for File-System Cache



11.3.3 Log-Structured File Systems

- With CPUs faster, memory larger
  - disk caches are also getting larger

  - increasing number of read requests come from file system cache
  - Thus, most disk accesses will be writes

- LFS treats the entire disk as a log
  - all writes are initially buffered in memory
  - periodically commit the writes to the end of the disk log
  - When file is opened, locate i-node, then find blocks

12 Raid

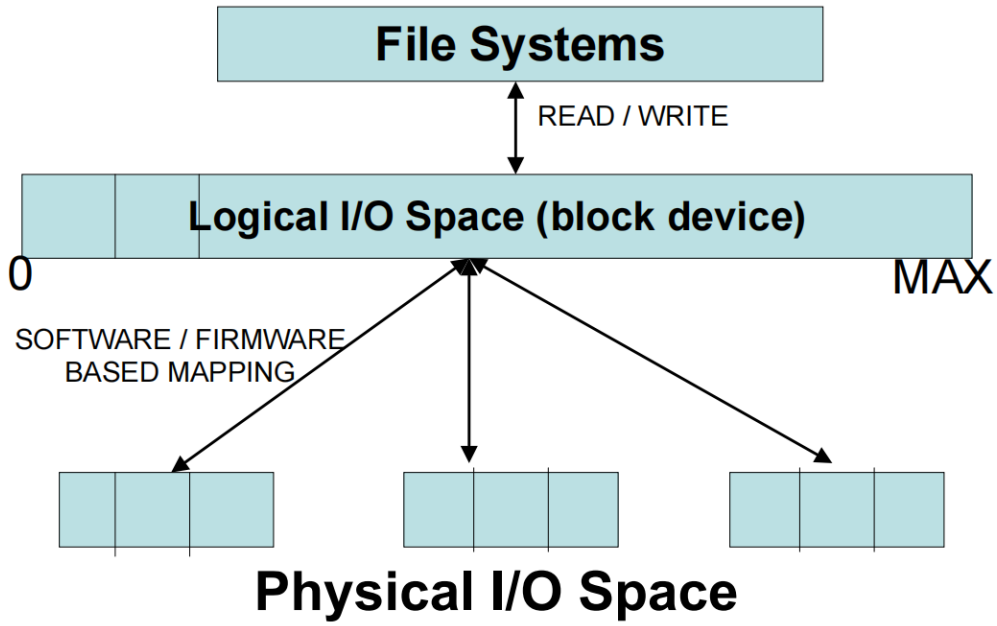
12.1 RAID — Original Motivation

- Replacing large and expensive mainframe hard drives (IBM 3310) by several cheaper Winchester disk drives
- Will work but introduces a data reliability problem:
  - Consider Mean Time To Failure (MTTF)
  - Assume MTTF of a disk drive is 30,000 hours
  - MTTF for a set of n drives is 30,000/n, n = 10 means MTTF of 3,000 hours

12.2 RAID — Today's Motivation

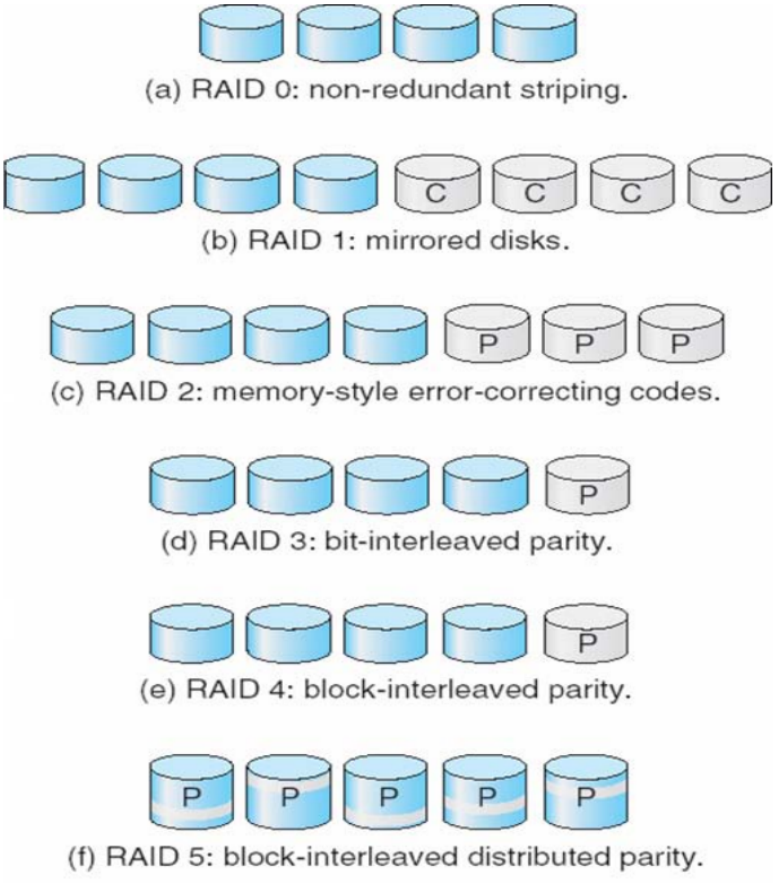
- "Cheap" hard drives are now big enough for most applications
- We use RAID today for
  - Increasing disk throughput(bandwidth) by allowing parallel access
  - Eliminating the need to make disk backups: Disks are too big to be backed up efficiently

12.3 Logical-to-Physical I/O Address Space Mapping



Logical I/O Space: virtual disk
Software/Firmware Based mapping: Raid

12.4 Several Levels of RAID



12.4.1 RAID 0

- Striping: Spread the data over multiple disk drives
- No fault tolerance
- But, much better I/O throughput: Number of I/O operations per second

DISK 0	DISK 1	DISK 2	DISK 3
BLOCK 0	BLOCK 1	BLOCK 2	BLOCK 3
BLOCK 4	BLOCK 5	BLOCK 6	BLOCK 7
BLOCK 8	BLOCK 9	BLOCK 10	BLOCK 11
BLOCK 12	BLOCK 13	BLOCK 14	BLOCK 15

12.4.2 RAID 1

- Mirroring: Two copies of each disk block
- Advantage:
  - Simple to implement
  - Fault-tolerant
- Requires twice the disk capacity

DISK 0	DISK 1	MIRROR 0	MIRROR 1
BLOCK 0	BLOCK 1	BLOCK 0	BLOCK 1
BLOCK 2	BLOCK 3	BLOCK 2	BLOCK 3
BLOCK 4	BLOCK 5	BLOCK 4	BLOCK 5
BLOCK 6	BLOCK 7	BLOCK 6	BLOCK 7

12.4.3 RAID 2

- Use an error (detection + correction) code instead of duplicating the data blocks
- Meant for disks that don't have built-in error detection.
- Modern disks support built-in error detection, so this level is mostly unused.

DISK 0	DISK 1	DISK 2	PARITY 1	PARITY 2
BLOCK 0	BLOCK 1	BLOCK 2	F(0,1,2)	
BLOCK 3	BLOCK 4	BLOCK 5	F(3,4,5)	
BLOCK 6	BLOCK 7	BLOCK 8	F(6,7,8)	
BLOCK 9	BLOCK 10	BLOCK 11	F(9,10,11)	

F = FUNCTION FOR ERROR DETECTION + CORRECTION

12.4.4 RAID 3

- N+1 disk drives: N data drives + 1 parity drive
- Data Block b[k] partitioned into N fragments b[k,1], b[k,2], ... b[k,N]
- Parity drive contains XOR (exclusive or) of these N fragments: p[k] = b[k,1] XOR b[k,2] XOR ... XOR b[k,N]
- Upon a failure, reconstruct the lost fragments by XOR of corresponding fragments from remaining drives. b[k,i] = p[k] XOR b[k,1] XOR ... b[k,i-1] XOR b[k,i+1] ... XOR b[k,N]
- Simple to implement in firmware/software
- Permits only one I/O operation at a time over entire array

DISK 0	DISK 1	DISK 2	PARITY
BLOCK [0,1]	BLOCK [0,2]	BLOCK [0,3]	PARITY 0
BLOCK [1,1]	BLOCK [1,2]	BLOCK [1,3]	PARITY 1
BLOCK [2,1]	BLOCK [2,2]	BLOCK [2,3]	PARITY 2
BLOCK [3,1]	BLOCK [3,2]	BLOCK [3,3]	PARITY 3

12.4.5 RAID 4

- Requires N+1 disk drives (as in RAID 3): N data drives + 1 Parity drive
- Data striped at block granularity (as in RAID 0): Disk 1 has block 1, disk 2 has block 2, and so on.

- Parity drive contains exclusive or of the N blocks in stripe: p[k] = b[Nk] XOR b[Nk+1] XOR ... XOR b[Nk+N-1]
- Multiple Read I/O operations can be processed in parallel
- But how about parallel writes I/O operations?

DISK 0	DISK 1	DISK 2	PARITY
BLOCK 0	BLOCK 1	BLOCK 2	PARITY 0
BLOCK 3	BLOCK 4	BLOCK 5	PARITY 1
BLOCK 6	BLOCK 7	BLOCK 8	PARITY 2
BLOCK 9	BLOCK 10	BLOCK 11	PARITY 3

12.4.6 RAID 5

- Single parity drive of RAID-4 is involved in every write
  - Will limit write parallelism
  - Exercises one parity disk more than others
- Solution in RAID-5: Distribute the parity blocks among all N+1 drives
- Up to N/2 parallel writes

DISK 0	DISK 1	DISK 2	DISK 3
BLOCK 0	BLOCK 1	BLOCK 2	PARITY 0
BLOCK 3	BLOCK 4	PARITY 1	BLOCK 5
BLOCK 6	PARITY 2	BLOCK 7	BLOCK 8
PARITY 3	BLOCK 9	BLOCK 10	BLOCK 11

12.5 The write problem

Every time a block is updated, the parity must be updated as well.

12.5.1 Naive Solution

- Assume we want to update the kth block bkold to bknew
- Before writing bknew: pold = b0old XOR b1old XOR ... bkold ... XOR bNold

- After writing bknew , we can naively recompute pnew as follows: pnew = b0old XOR b1old XOR ... bknew ... XOR bNold
- The overhead is high
  - N-1 reads: Read all old data blocks except the block being written (bknew)
  - 2 writes: bknew and pnew

12.5.2 Smarter Solution

- Assume we want to update the kth block bkold to bknew
- Before writing bknew: (A) pold = b0old XOR b1old XOR ... bkold ... XOR bNold
- Moving bkold to left hand side: (B) pold XOR bkold = b0old XOR b1old XOR ... XOR bNold
- Naive solution to compute pnew: (C) pnew = b0old XOR b1old XOR ... bknew ... XOR bNold
- Moving bknew to left hand side: (D) pnew XOR bknew = b0old XOR b1old XOR ... XOR bNold
- Combining (B) and (D): (E) pnew XOR bknew = pold XOR bkold
- Smarter solution: moving bknew to right hand side: pnew = bknew XOR pold XOR bkold
  - 2 reads: bkold and pold, Or just one read of pold if bkold was read into memory earlier
  - 2 writes: bknew and pnew

12.6 Comparasion

	RAID 0 (N disks)	RAID 1 (N disks)	RAID 2 (N+1) disks	RAID 3 (N+1) disks	RAID 4 (N+1) disks	RAID 5 (N+1) disks
Fault-tolerance	None	All 1-disk and most 2-disk failures	1-disk failure with error detection and correction	1-disk failure with Error Correction	1-disk failure with Error Correction	1-disk failure with Error Correction
Max. READ Parallelism	N	N	N	1 (none)	N	N+1
Max. WRITE Parallelism	N	N/2	1 (none)	1 (none)	1 (none)	(N+1)/2
Space Overhead	0%	100%	(k/N)x100% for K parity disks	(1/N)x100%	(1/N)x100%	(1/N)x100%

12.7 Conclusion

- RAID original purpose was to take advantage of commodity drives that were smaller and cheaper than conventional disk drives: Replace a single large drive by an array of smaller drives
- Nobody does that anymore!
- Today: Main purpose of RAID is to build fault-tolerant storage systems that do not need backups and deliver high throughput.
- Low cost of disk drives makes RAID-1 attractive for small installations, for we have now very cheap RAID controllers
- Otherwise prefer
  - RAID-3 for simplicity
  - RAID-5 for higher parallelism
- Often combined with NVRAM to improve write performance

13 Introduction to Virtual Machines

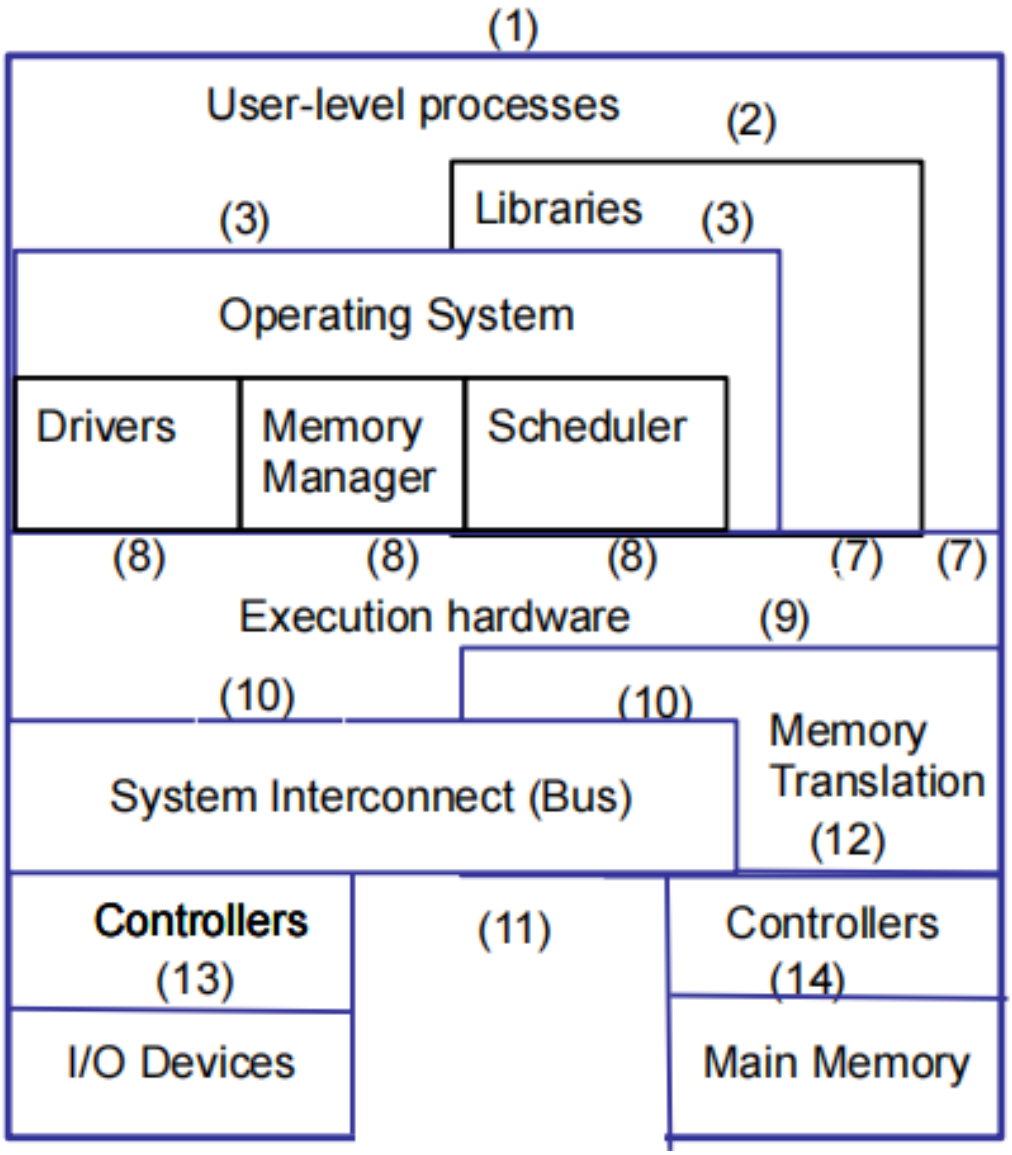
13.1 Virtualization

- Makes a real system appear to be a set of virtual systems
- One-to-many virtualization
  - E.g. one physical machine may appear as multiple virtual Machines
  - one physical memory to many virtual memory
  - one physical CPU to many virtual CPU
  - one physical disk may look like multiple virtual disk
  - one physical network may look like multiple virtual networks
- Many-to-one virtualization: Raid. Many physical machines/disks/networks may appear to look like one virtual machine/disk/network etc
- Many-to-many virtualization: Extend the above statements

13.2 Virtual Machines

- Logical/Emulated representations of full computing system environment
  - CPU + memory + I/O
  - Implemented by adding layers of software to the real machine to support the desired VM architecture.
- Uses:
  - Multiple OSes on one machine, including legacy OSes
  - Isolation
  - Enhanced security
  - Live migration of servers
  - Virtual environment for testing and development
  - Platform emulation
  - On-the-fly optimization
  - Realizing ISAs not found in physical machines

13.3 Interfaces of a computer system



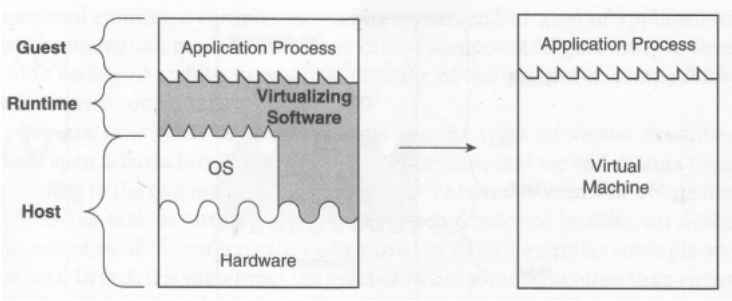
- User ISA: 7
- System ISA: 8
- Syscalls: 3

- ABI: 3, 7
- API: 2, 7

13.4 Two Types of VMs

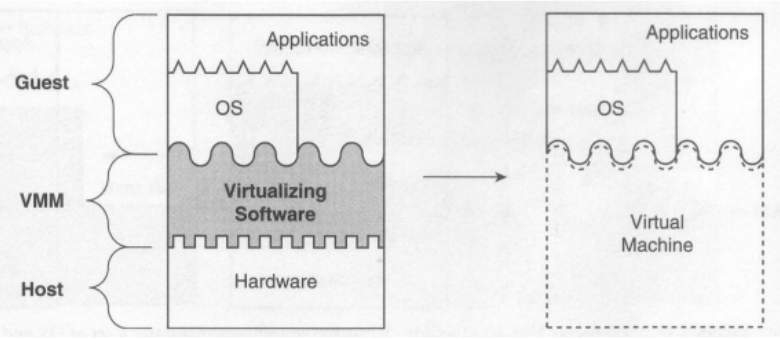
13.4.1 Process VM

- Virtualizes the ABI
- Virtualization software = Runtime
- Runs in non-privileged mode (user space)
- Performs binary translation.
- Terminates when guest process terminates.



13.4.2 System VM

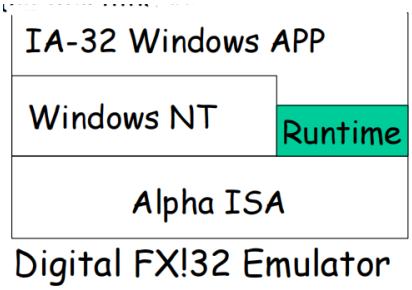
- Virtualizes the ISA
- Virtualization software = Hypervisor
- Runs in privileged mode
- Traps and emulates privileged instructions

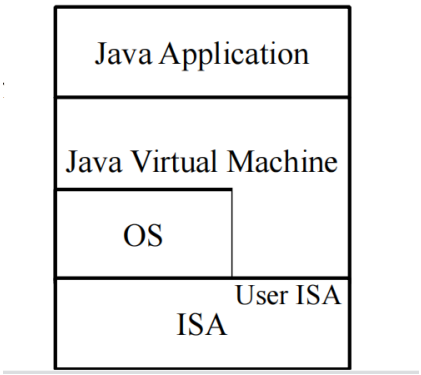


13.5 Process Virtual Machines

- Process in a multiprogramming OS
  - Standard OS syscall interface + instruction set
  - Multiple processes, each with its own address space and virtual machine view.
- Emulators
  - Support one ISA on hardware designed for another ISA
  - Interpreter: Fetches, decodes and emulates individual instructions. Slow. like Python.
  - Dynamic Binary Translator:
    - Blocks of source instructions converted to target instructions.
    - Translated blocks cached to exploit locality.
- Same ISA Binary Optimizers
  - Optimize code on the fly
  - Same as emulators except source and target ISAs are the same.
- High-Level Language VMs
  - Virtual ISA (bytecode) designed for platform independence
  - Platform-dependent VM executes virtual ISA

- E.g. Sun's JVM and Microsoft's CLI (part of .NET)
- Both are stack-based VMs that run on register-based m/c



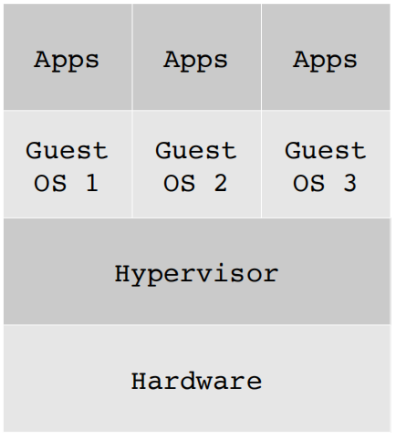


13.6 System Virtual Machines

13.6.1 Hypervisor

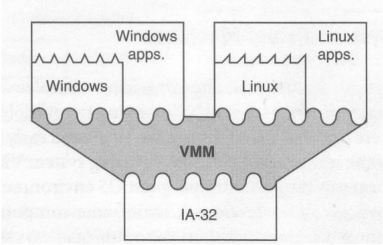
- Also called Virtual Machine Monitor (VMM)
- A hypervisor is an operating system for operating systemsL I/O, multiplexing, protection
  - Key difference: Hypervisor is always transparant, it shouldn't let OS know if it's running on a hardware or a virtual machine
  - Provides a virtual execution environment for an entire OS and its applications
  - Controls access to hardware resources

– Key feature: When guest OS executes a privileged instruction, Hypervisor intercepts the instruction, checks for correctness and emulates the instruction.



13.6.2 Type 1 Hypervisors(Classical System VMs)

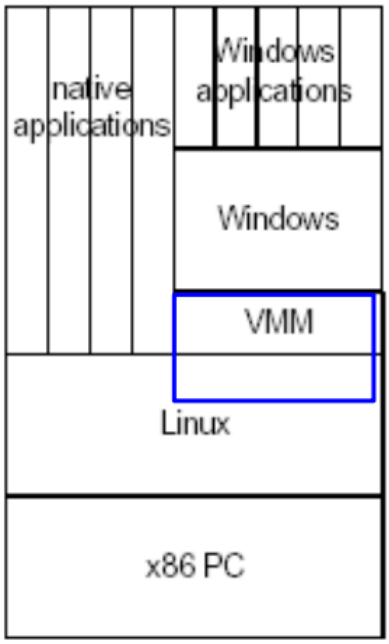
- Hypervisor executes natively on the host ISA
- Hypervisor directly controls hardware and provides all device drivers
- Hypervisor emulates sensitive instructions executed by the Guest OS
- E.g. KVM(questionable) and VMWare ESX Server
- Disadvantage: Big and Complex with device drivers



13.6.3 Type 2 Hypervisors (Hosted VMs)

- A host OS controls the hardware

- The Hypervisor runs partly in process space and partly in the host kernel
- Hypervisor Relies on host OS to provide drivers
- E.g. VMWare Desktop Client, KVM
- Disadvantage: inherited all the overheads of host OS



13.6.4 Para-virtualized VMs(can be type1 or type2)

- Modify guest OS for better performance: keep hypervisor transparant for most purposes, but for some performance sensitive reasons we make hypervisor visible by hyper calls to do I/O or memory virtualization.
- Traditional Hypervisors provide full-virtualization
  - They expose to VMs virtual hardware that is functionally identical to the underlying physical hardware.
  - Advantage : allows unmodified guest OS to execute

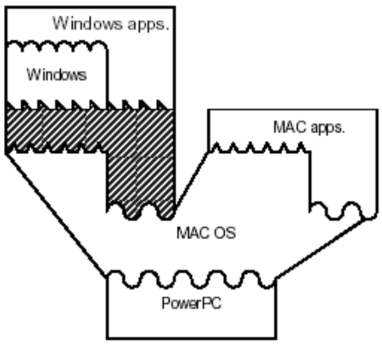
  - Disadvantage: Sensitive instructions must be trapped and emulated by Hypervisor.
  - E.g. KVM and VMWare ESX provide full virtualization

- Para-virtualized VM
  - Sees a virtual hardware abstraction that is similar, but not identical to the real hardware.
  - Guest OS is modified to replace sensitive instructions with "hypercalls" to the Hypervisor.
  - Advantage: Results in lower performance overhead
  - Disadvantage: Needs modification to the guest OS.
  - E.g. Xen provides both para-virtual as well as full-virtualization

- Often traditional Hypervisors are partially para-virtualizated: Device drivers in guest OS may be para-virtualized whereas CPU and Memory may be fully virtualized

13.6.5 Whole System VMs: Emulation

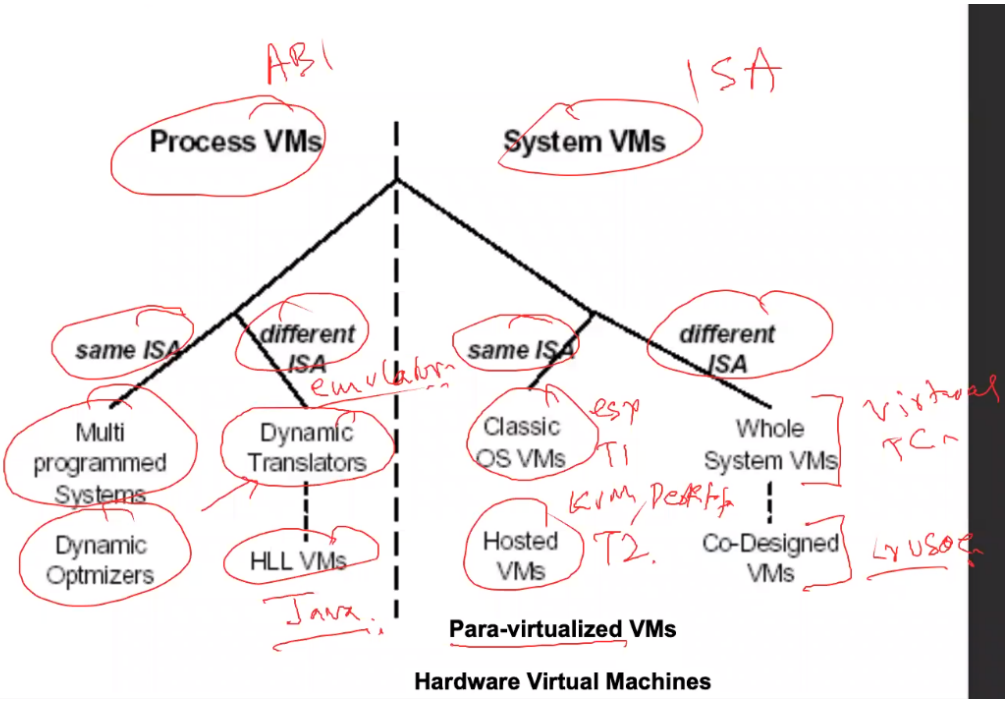
- Host and Guest ISA are different
- So emulation is required
- Hosted VM + emulation
- E.g. Virtual PC (Windows on MAC)



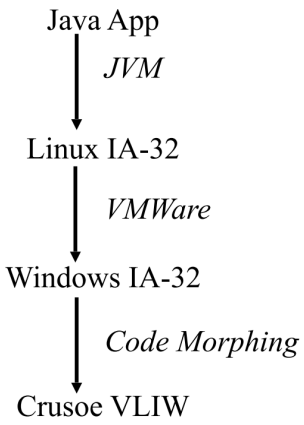
13.6.6 Co-designed VMs

- The hypervisor is designed closely with (and possibly built into) a specific type of hardware ISA (or native ISA).
- Goal: Performance improvement of existing ISA (or guest ISA) during runtime.
- Hypervisor performs Emulation from Guest ISA to Native ISA.
- E.g. Transmeta Crusoe
  - Native ISA based on VLIW
  - Guest ISA = x86
  - Goal power savings

13.7 Taxonomy



13.8 Versatility



13.9 Virtualizing individual resources in System VMs

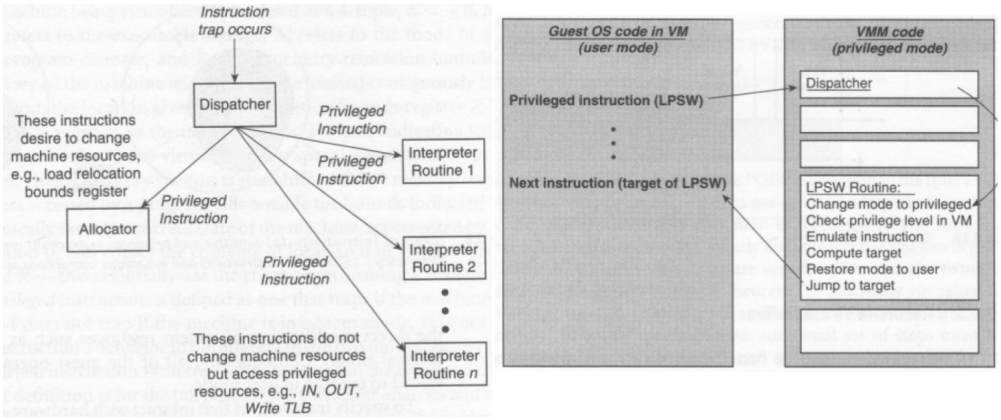
13.9.1 CPU Virtualization for VMs

Host VM(KVM)

- Each VM sees a set of "virtual CPUs"
- QEMU(a user process) would create the guest OS, then make a regular place within process whenever create as many threads as the number of virtual cpu. Each of the threads would become a virtual cpu for the guest OS.
  - these threads switch between guest mode and root mode, and it's the most cost for virtualization
  - guest mode → root mode: VM exit, it's a trap
  - root mode → guest mode: VM entry
- Hypervisors must emulate privileged instructions issued by guest OS.
- Modern ISAs provide special interfaces for Hypervisors to run VMs
  - Intel provides the VTx interface
  - AMD provides the AMD-v interface

- These special ISA interfaces allow the Hypervisors to efficiently emulate privileged instructions executed by the guest OS.
- When guest OS executes a privileged instruction
  - Hardware traps the instruction to the hypervisor
  - Hypervisor checks whether instruction must be emulated.
  - If so, Hypervisor reproduces the effect of privileged operation.

13.9.2 Execution of Privileged Instruction by Guest



Dispatcher: the enter point for hypervisor

13.9.3 Resource Control

- Issue: How to retain control of resources in the Hypervisor?
- Interrupt: Timer(LAPIC) interval control performed by Hypervisor
- Also, guest OS is not allowed to read the timer value: Guest OS sees a virtual interval timer
- Hypervisor also gains control whenever guest OS executes privileged instructions

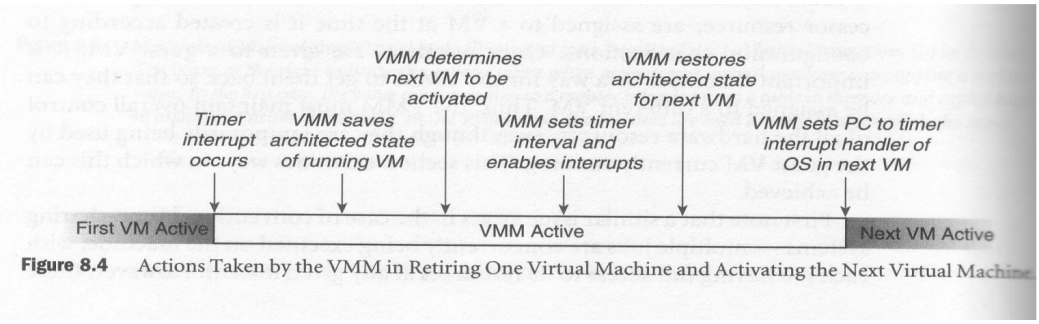
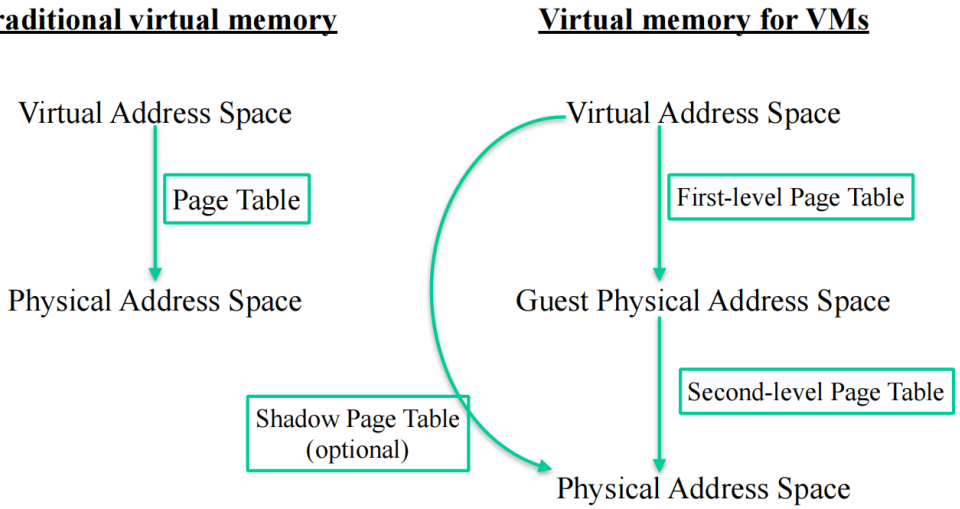


Figure 8.4 Actions Taken by the VMM in Retiring One Virtual Machine and Activating the Next Virtual Machine

13.9.4 Memory Virtualization for VMs



- Guest OS in each VM sees a "guest"-physical address (GPA) space instead of the physical addresses
- Often hardware supports two-level page tables: EPT in Intel VT-x and NPT in AMD-v
- When hardware doesn't, then Hypervisor needs to emulate two-level page tables using "shadow page tables".

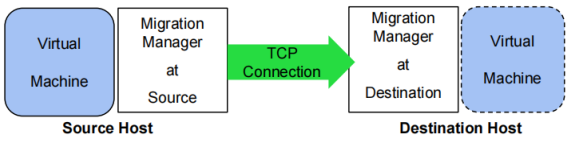
13.9.5 I/O Virtualization for VMs

- Hypervisor provides a virtual version of each physical device

- I/O activity directed at the virtual device is trapped by Hypervisor and converted to equivalent request for the physical device.
- Options:
  - Device emulation
    - Hypervisor traps and emulates each I/O instruction from Guest in Hypervisor.
    - Very slow.
    - Difficult to emulate the effect of combinations of I/O instructions
  - Para-virtual devices
    - Special device drivers inserted in guest OS to talk to Hypervisor.
    - Most common
  - Direct device access
    - Allow the VM to directly access physical device.
    - Fastest option but not scalable.
    - Requires IOMMU(protection) and VT-d support from hardware

14 Live Migration of Virtual Machines

14.1 What is live VM migration?



- Move a VM from one physical machine to another even as its applications continue to execute during migration
- Live VM migration usually involves
  - Migrating memory state

  - Migrating CPU state
  - Optionally, migrating virtual disk state

- Migration managers at source and destination
  - Connect via TCP connection
  - At source, the migration manager maps the guest VM's memory and execution state
  - Transfers VM's pages to the target migration manager over TCP connection.
  - At destination, the migration manager restores the VM's state and resumes execution
  - Migration manager examples: xend for Xen, QEMU for KVM

14.2 Why Live VM Migration?

Why Migrate?

- Load Balancing: Move VMs from highly loaded servers to lightly loaded servers
- Server maintenance: When server needs to be upgraded
- Energy savings: Move out VMs before shutting down servers to reduce energy usage

Why live?

- To keep long-running jobs alive
- To keep network connections alive
- Broadly, to avoid disruptions to users of VM

Why VM?

- Why not migrate individual processes?
- Process migration is very messy, and may leave residual dependencies (state) at source host. E.g. system call redirection, shared memory, open files, inter-process communication, etc

14.3 Performance Goals in Live Migration

- Minimizing Downtime
- Reducing total migration time
- Avoiding interference with normal system activity
- Minimizing network activity

14.4 Migrating Memory

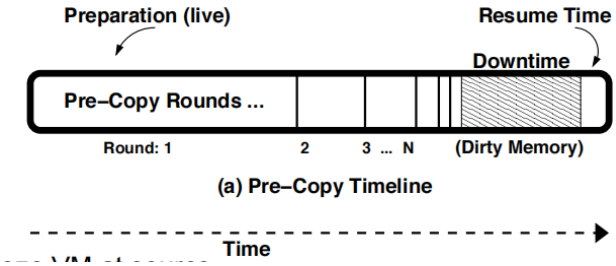
14.4.1 Pure stop-and-copy/Non-live migration

- Freeze VM at source,
- Copy the VM's pseudo-physical memory contents to target,
- Restart VM at target
- Longest downtime.
- Fastest migration: Minimal total migration time = downtime

14.4.2 Pure Demand Paging

- Freeze VM at source,
- Copy minimal execution context to target: PC, Registers, non-pageable memory
- Restart VM at target,
- Pull memory contents form source as and when needed by destination hypervisor and source hypervisor
- Smaller downtime
- Sloooow warm-up phase at target during page-faults across network

14.4.3 Pre-copy migration



- DON'T freeze VM at source: Let the VM continue to run
- Copy VM's pseudo-physical memory contents to target over multiple iterations
  - First iteration → copy all pages.
  - Each subsequent iteration → copy pages that were dirtied by the VM during the previous iteration
- Do a short stop-and-copy when number of dirty pages is "small enough".
- But what if number of dirty pages never converges to a small enough number? After a fixed number of iterations, give up and stop-and-copy
- Not good for write-intensive applications

So what's the catch? How do we track dirtied pages?

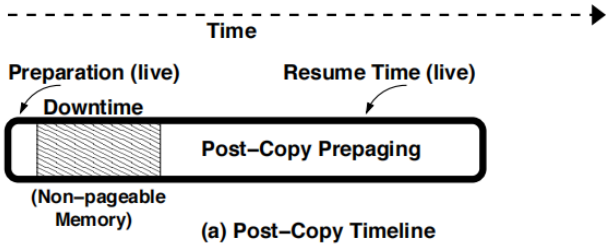
- Mark the VM's memory pages as read-only after each iteration.
- Trap write operations via hypervisor to xend and track dirtied pages.
- Reset after each iteration
- Works well as long as writes are infrequent

Optimizations

- Limit the bandwidth used by migration: To minimize impact on running services
- Stun Rogue Processes: Those that don't stop dirtying memory

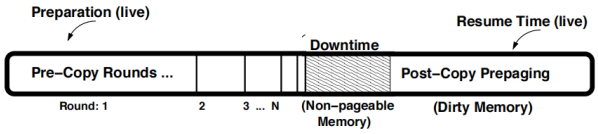
- Free Page Cache Pages
  - Can be re-cached at target
  - Potential performance hit

14.4.4 Post-copy migration



- Freeze the VM first
- Migrate CPU state and minimum state to destination
- Start VM at the target, but without its memory!
- Transfer memory by concurrently doing the following
  - Demand paging over network
  - Actively pushing from source
  - Hopefully most pages will be pushed BEFORE they are demand paged.
- Advantage:
  - Each page transferred over the network only once.
  - Deterministic total migration time
- Disadvantage:
  - Cold start penalty at the destination
  - If migration fails, then VM is lost

14.4.5 Hybrid pre/post-copy



Combines the benefits and drawbacks of both

1. Perform one or more rounds of live pre-copy rounds
2. Pause VM and transfer execution state
3. Use post-copy to transfer any remaining dirty pages from source

14.5 Migrating Network Connections

14.5.1 Within a LAN

- the migrated VM carries its IP address, MAC address, and all protocol state, including any open sockets
- Backward (re)learning(Transparant bridging) delay at the network switches
  - Switches needs to re-learn the new location of migrated VMs MAC address
  - Solution: Send an unsolicited ARP reply from the target host.
  - Intermediate switches will re-learn automatically.
  - Few in-flight packets might get lost

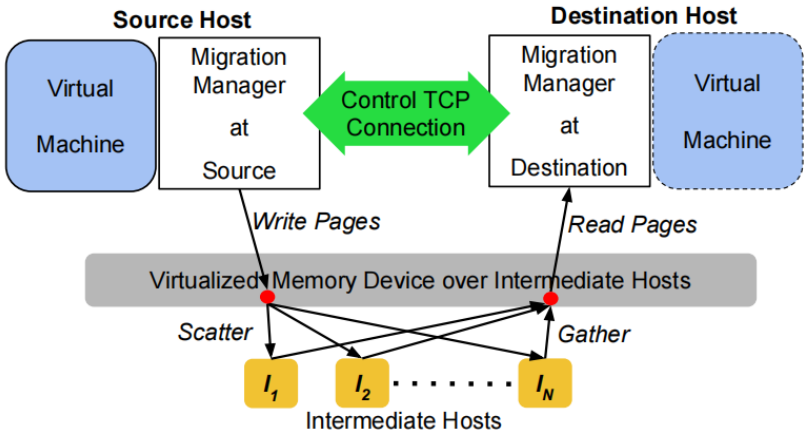
14.5.2 Across a WAN

- Source and destination subnets may have different IP addresses.
- Active network connections may need to be tunneled via VPN or similar mechanisms

14.6 Storage Migration

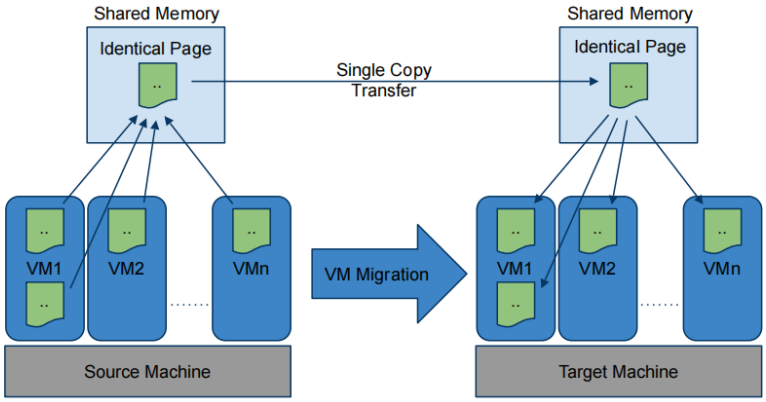
- Many gigabytes of local disk image possible.
- For LAN
  - Assume the storage is over the network and remains accessible from the new target machine.
  - E.g. Network File System (NFS), or Network Block Device(NBD), or iSCSI etc.
- For WAN
  - Disk image may need to be transferred.
  - Can use pre-copy or post-copy for disk images
  - Combined bandwidth saving optimizations such as compression, and/or de-duplication.

14.7 Scatter-Gather migration



Scatter-Gather migration: The VM's state is transferred through intermediaries. A direct connection between the source and destination carries control information, faulted pages, and some actively pushed pages

14.8 Multi-VM (Gang) Migration



- De-duplicate memory pages to reduce network traffic.
- Identify identical pages across multiple VMs: By comparing byte-wise (expensive), or checksum (cheaper)
- Send only one copy of identical page to destination node
- Destination Node replicates the pages to multiple VMs

15 Operating Systemand Security

15.1 What is Security

Goal	Threat
Data confidentiality	Exposure of data
Data integrity	Tampering with data
System availability	Denial of service

Preventing unauthorized users from executing undesirable actions, such as

- Stealing your data (C)
- Giving you fake data/Tampering your data (I)
- Preventing you from doing your work (A)

15.2 Securing what? From what?

- Securing the OS from users: OS-level mechanisms
- Securing one user from another: Access control, isolation
- Securing users from OS! Yes, sometimes the OS is not trusted by the user. E.g. in a cloud users may not trust the cloud platform's OS.

15.3 Security mechanisms in OS and hardware

- CPU Execution privileges ("Who can access?")
  - Part of CPU state
  - x86 privilege rings (0,1,2,3) in EFLAGS
  - VTx provides root and non-root modes for virtualization because 0123 is not enough for hypervisor. So hypervisor has a root-mode 0123.
- Memory protection ("What can be accessed?")
  - Protection bits in segment descriptors
  - Protection bits in page-table registers
  - Virtual Memory (naming)
- File system privileges ("What can be accessed?")
  - User accounts
  - Access permissions

15.4 Common Motivations of Intruders

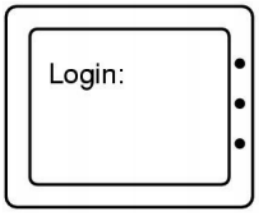
1. Peeping Tom: Casual prying by nontechnical users
2. Insider threat: Disgruntled insiders, programmers backdoor
3. Extortion: Make money
4. Espionage/Intelligence gathering: Commercial or military or government

5. Hacktivism: Political or social motivation
6. Sometimes motivations may overlap: Was Snowden incident 2? 4? 5? All?

15.5 User Authentication

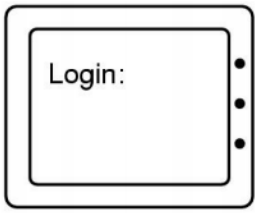
- Verifying that you are who you claim you are.
- File permissions and user's rights are set according to user's identity, which is established by authentication.
- Basic Principles. Authentication must identify:
  - Something the user knows
  - Something the user has
  - Something the user is
- This is done before user can use the system

15.6 Login Spoofing



(a)

Correct login Screen



(b)

Phony login Screen

Countermeasures:

- Careful user can intentionally enter a fake password the first (few) time(s).
- Use "Trusted Path"
  - A sequence of user actions that is guaranteed to give control to the OS

  - E.g. pressing Ctrl-Alt-Del could guarantee that legitimate login (or logout) screen will show up

15.7 Buffer Overflow

15.8 Memory reuse — Dumpster Diving

- Request memory, disk space, tapes
- Don't write. Just read and interpret existing data.
- May find passwords, ssh keys, emails, personal information, browsing history, etc
- CM:
  - Scrub memory/storage before allocating to user.
  - Encrypt data. Throw away the key once done.
  - Disadvantage: Takes more time

15.9 Logging

- Logs: A time-wise record of system activity. Events always appended. "Never" erased.
- Logs must be analyzed often to detect suspect activity
- What to log?
  - Too much logging
    - takes up storage
    - slows down normal operations.
    - Slows down analysis.
  - Too little logging and you miss critical events.
- Privacy risk
  - Can break laws.
  - Or violate user's perception of privacy. (sometimes more important.)

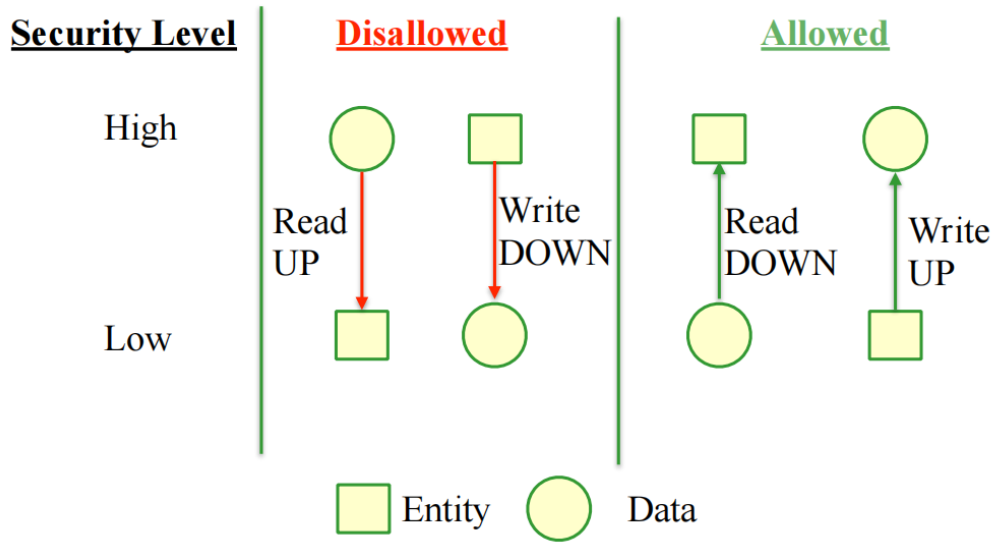
15.10 Access control

- Discretionary access control (DAC)
  - "John can access X. Alice can do Y."
  - Commodity systems
- Mandatory access control (MAC) (Multi-level Security)
  - Military/spy systems
  - More later
- Role-based access control (RBAC)
  - "CEO can do X. Software Engineer can do Y. Secretary can do Z".
  - Enterprise systems
- Administrative Role-based Access Control. "Dean can allow department chair to do X. Dept chair can allow secretary to do Y"

15.11 Multi-level Security

- Also called Mandatory Access Control (MAC): As opposed to Discretionary Access Control (DAC) in commodity systems.
- Data objects are classified at different levels
  - Top secret, secret, confidential, unclassified etc
  - Sometimes additional compartments: Crypto, Subs, NoForn
- People (and computers) have clearances
- Informally: To see a data object, you must have clearance for that level and for that compartment

15.11.1 No Read UP, No Write DOWN



15.11.2 MLS Pump

- In practice, to get things done, upper-level must at least acknowledge the receipt of data from lower level, But acks create a backdoor for covert channels (surreptitious communication)
- An MLS Pump
  - Allows acks from higher to lower levels,
  - but at such a low data rate that covert channels become impractical

16 I/O Models