Operating System

Pham Minh Tuan - 1752595

Contents:

1 Overview

"What is Operating System ?"

- An operating system acts an intermediary(trung gian) between the user of a computer and the computer hardware.
- Operating system provide an environment in which a user can execute programs in a *convenient* and *efficient* manner.
- An Operating System is software that manages the computer hardware, the hardware must provide appropriate mechanisms to ensure the correct operation and prevent user from interfering with the proper operation of the system.

1.1 What Operating System can do ?

1. A computer system can be divided into 4 components: the **hardware**, the **operating system**, the **application programs**, and the **users**.
   - The Hardware includes the **central unit processing** (CPU), the **memory**, and the **input-output** (IO) **devices**
   - Hardware provides the basic computing resources for the system.
   - The Application programs define the ways in which these resources are used to solve user's computing problem.
   - The operating system controls the hardware and coordinates it use among the various application programs for various users.
   - In system's view, the operating system is the program most intimately involved with the hardware. an operating system can be viewed as **resource allocator**.
   - An operating system is a control program that manages the execution of user programs to prevent errors and improper use of the computer.
   - An operating system is a program running at all times on the computer - usually called the **kernel**
2. **Computer-System Operation**
   - For a computer to start running, it needs to have an initial program to run or **bootstrap program**.
   - Bootstrap program is stored in **ROM** (read-only memory) or **EEPROM** (electrically erasable programmable read-only memory) or **firmware**.
   - Bootstrap program initializes all aspects of the system, it knows how to load the operating system and start executing that system.
   - The operating system then starts executing the first process and waits for the occurrence events.
   - The occurrence of an event is usually signaled by an **interrupt** from either software or hardware.
     - Hardware may trigger an interrupt at any time by sending a signal to the CPU.
     - Software may trigger a special operation called a **system call** or **monitor call**.
3. **Interrupt:**
   - When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location.

- The fixed location usually contains the starting address where service routine for the interrupt is located.
- The interrupt must transfer control to the appropriate interrupt service routine.
- The **interrupt vector** provide the address of the interrupt service routine for the interrupting device.
- The interrupt architecture must also save the address of the interrupted instruction.
- Incoming interrupts are disabled while another interrupt is being processed to prevent a **lost interrupt**
- A **trap** is a software-generated interrupt caused either by an error or a user request.
- An operating system is **interrupt driven**.

4. **Storage Structure**

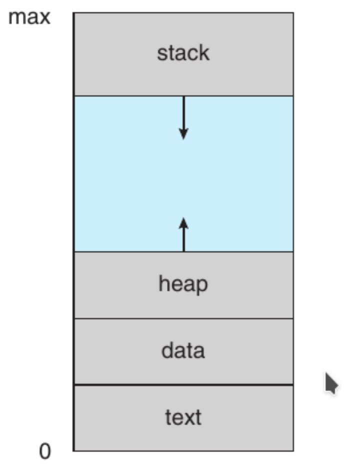
- General-purpose computers run most of their program from rewritable memory, called main memory (or random-access memory (RAM)).
- Main memory is commonly implemented in a semiconductor technology called **dynamic random-access memory** (DRAM).
- Main memory is usually too small to store all needed programs and data permanently.
- Main memory is a *volatile* storage device that loses its contents when power is turned off or otherwise lost. $\rightarrow$ **secondary storage** for larges quantities of data permanently.
- **magnetic disk**: provide storage for both program and data.
- Most programs (system and application) are storage on a disk until they are loaded into memory.
- Differences between among the various storage systems lie on speed, cost, size and volatility.

- In the absence of power and general backup systems, some data must be written to **nonvolatile storage**.
- Some nonvolatile disks are electronic disk, NVRAM (DRAM with battery backup power).

2 Chapter 3: Process

2.1 Process Concept

1. **The Process**
   - The process:
     - A process is a program in execution.
     - The status of the current activity of a process is represented by the value of the **program counter** and the contents of the processor's register.
     - The memory layout of a process is typically divided into multiple sections
       * **Text section:** - the executable code
       * **Data section:** - global variables
       * **Heap section:** - memory that is dynamically allocated duirng program run time.
       * **Stack section:** - temporary data storage when invoking functions (parameters, return val, ...



etc).

     - The size of text and data section is **fixed**, while the stack and heap section can shrink and grow **dynamically** during program execution.
     - the stack and heap sections grow **toward** one another, the operating system must ensure they do not **overlap** one another.
     - A program itself is not a process, which is called **passive entity** while process is called **active** entity with a program counter specifying the next instruction to execute and a set of associated resources.
     - A program become a process when when an executable file is loaded into a memory.
     - Two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.
     - A process can itself be an execution environment for another code. For instance, the JVM

2. **Process State**
   - The state of the process is defined in part of the current activity of that process.
     - **New**: The process is being created.
     - **Running**: Instructions are being executed.

- **Waiting**: The process is waiting for some events to occur (such as I/O completion or reception of a signal).
- **Ready**: The process is waiting to be assigned to a processor.
- **Terminated**: The process has finished execution.

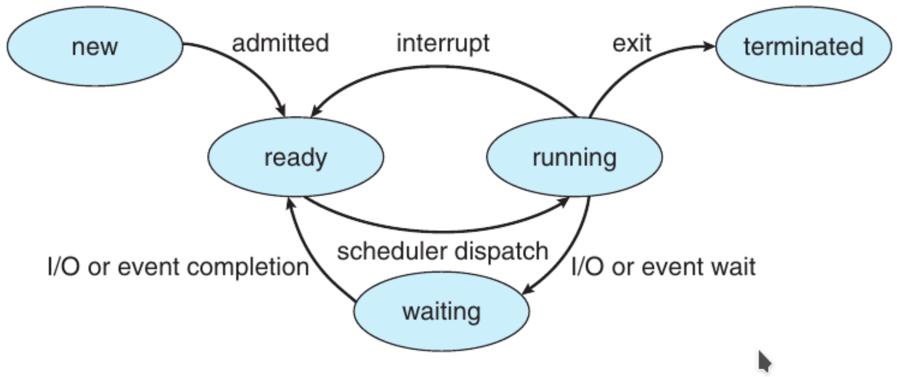


Figure 3.2 Diagram of process state.

- Only **one** process can be *running* on any processor core at any instant. Many process may be *ready* and *waiting*.

3. **Process Control Block**

   - Each process is represented in the operating system by a **process control block (PCB)** or **task control block**.
     - Process state
     - Program counter
     - CPU register
     - CPU-scheduling information
     - Memory-management information
     - Accounting information
     - I/O status information

4. **Threads**

   - A process is a program that performs a single **thread** of execution.

2.2 Process Scheduling

- The objective of multiprogramming is to have some process running at all the times so as to maximize CPU utilization.
- The objective for time sharing is to switch a CPU core among processes so frequently that users can interact with each program while it is running.
- To meet those objective, **process scheduler** is used to select an available process for program execution on a core.

- The number of processes currently in memory is known as the **degree of multiprogramming**.
- Most processes can be described as either I/O bound or CPU bound.
  - An **I/O bound process** is one that spends more of its time doing I/O than it spends doing computations.
  - A **CPU-bound process** generates I/O requests infrequently, using more of its time doing computations.

1. **Scheduling Queues**
   - As a processes enter the system, they are put into the **ready queue**, where they are ready and waiting to execute on a CPU's core. This queue is a linked-list.
   - When a process is allocated a CPU core, it executes for a while and eventually terminates, is interrupted, or waits for the occurrence of a particular event - I/O request. Therefore, the process has to wait for the completion of I/O - which is placed in a **wait queue**.

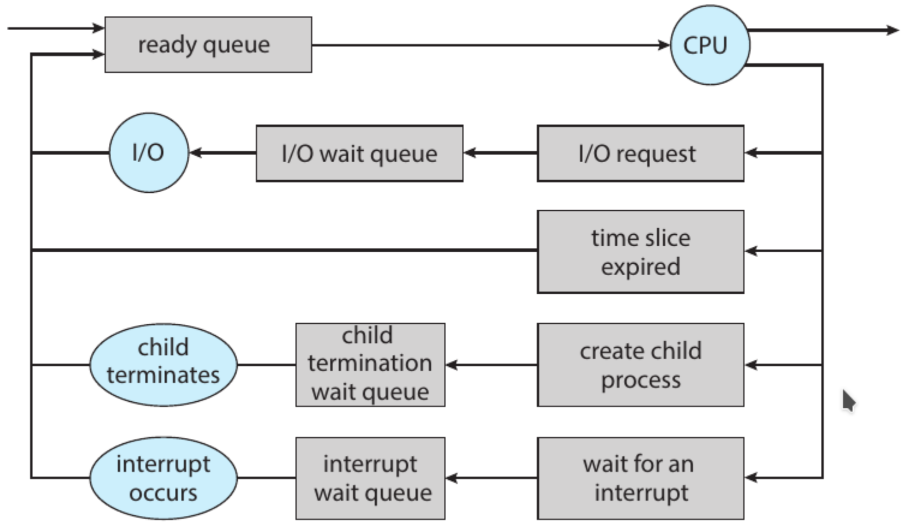


Figure 3.5 Queueing-diagram representation of process scheduling.

2. **CPU Scheduling**
   - The role of **CPU scheduler** is to select from among the processes that are in the ready queue and allocate a CPU core to one of them.
   - The scheduler must select a new process for the CPU frequently.

2.3 Schedulers

- **Long-term scheduler** (or **job scheduler**):
  - Selects which processes should be brought into the ready queue.
  - It is invoked very **frequently** (milliseconds) $\rightarrow$ fast.

- **Short-term scheduler** (or **CPU scheduler**):
  - Selects which process should be executed next and allocates CPU.
  - It is invoked very **infrequently** (seconds, minutes) $\rightarrow$ slow.
- The long-term scheduler controls the **degree of multiprogramming**.
- Processes can be described as:
  - **I/O-bound process**: spends more time doing I/O than computations, many short CPU bursts.
  - **CPU-bound process**: spends more time doing computations than I/O; few very long CPU bursts.

2.4 Context Switch

- When a CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via **context switch**.
- **Context** of a process represented in the PCB.

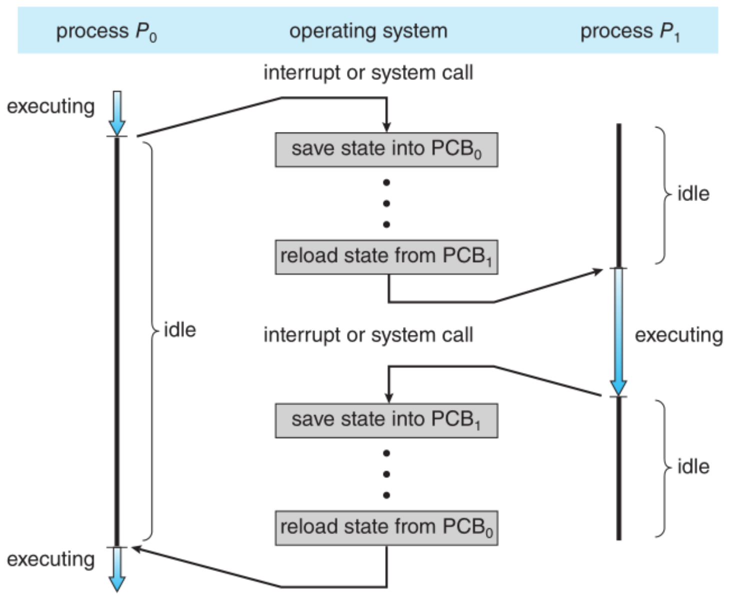
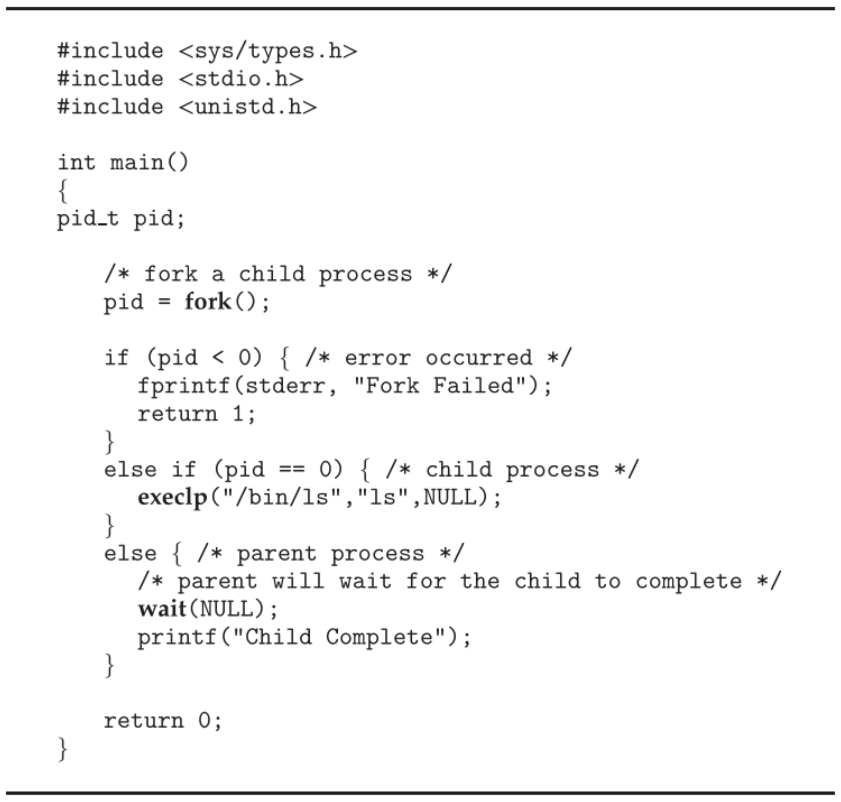


Figure 3.6 Diagram showing context switch from process to process.

2.5 Process Creation

- **Parent** process create **children** processes, which in turn create other processes $\rightarrow$ formming a tree of process :).
- Generally process is identified via **a process identifier** (PID).

- Resource sharing:
  - Parent and children share all resources.
  - Children share subset of parent's resources.
- Execution:
  - Parent and children execute concurrently.
  - Parent waits until child terminates.
- Address space of child process duplicate of parent; child has a program load into it.
- **fork()** system call creates new process which is an exact duplicate of the calling process at the time of its creation.



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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Figure 3.8 Creating a separate process using the UNIX fork() system call.

- **exec()** system call used after a **fork()** system call to replace the process's memory space with the new program.

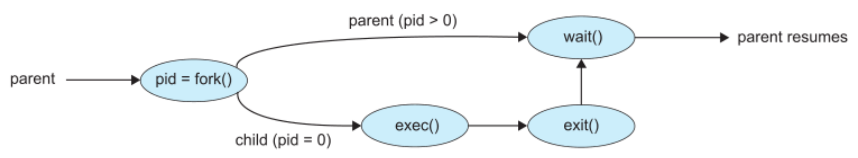


Figure 3.9 Process creation using the `fork()` system call.

2.6 Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the **exit()** system call.
- At that point, the process may return a status value (typically an integer) to its waiting parent process (via the wait() system call).
- All the resources of the process including physical and virtual memory, open files, and I/O buffers—are de-allocated and reclaimed by the operating system.
- Parent may terminate execution of children processes if:
  - Child has exceed allocated resources.
  - Task assigned to child is no longer required.
  - if parent is exiting $\rightarrow allchildrenterminated$ (**cascading termination**).

2.7 Interprocess Communication

- Processes within a system may be **independent** or **cooperating**.
- Cooperating process can affect or be affected by other processes, including sharing data.
- Reasons for cooperating processes:
  - Information sharing.
  - Computation speedup.
  - Modularity.
  - Convenience.
- **Independent** process **cannot** affect or be affected by the execution of another process.
- **Cooperating** process **can** affect or be affected by the another process.

End

3 Threads

- A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter (PC), a register set, and a stack.
- It shares with other threads belonging to the same process:
  - code section.
  - data section.
  - other operating system resources, such as open files and signals.
  - A traditional process has a single thread of control; If a process has multiple threads of control, it can perform more than one task at a time.

3.1 Motivation

- Threads run within application.
- Multiple tasks with the application can be implemented by separate threads:
  - Update display.
  - Fetch data.
  - Spell checking
  - Answer a network request.
- Process creation is **heavy-weight** while thread creation is **light-weight**
- Kernels are generally multithreaded.

3.2 Multicore Programming

- Multicore systems put pressure on programmers, challenges:
  - Dividing activities.
  - Balance.
  - Data splitting.
  - Data dependency.
  - Testing and debugging.

3.3 User Threads

- Thread management is done by user-level threads library.
- Three primary thread libraries:
  - POSIX **Pthreads**.
  - Win32 threads.
  - Java threads.

3.4 Kernel Threads

- Supported by kernel.
- Window, Solaris, Linux, MAC OS, etc.

3.5 Multithreading Models

- Many-to-One (Solaris, GNU)
- One-to-One (Window, Linux)
- Many-to-Many. (Solaris 9, Window NT/2000)

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  /* set the default attributes of the thread */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid, &attr, runner, argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

3.6 Linux Threads

- Thread creation is done through **clone()** system call.

- **clone()** allows a child task to share the address space of the parent task (process).
- **struct** `task_struct` points to process data structures (shared or unique)

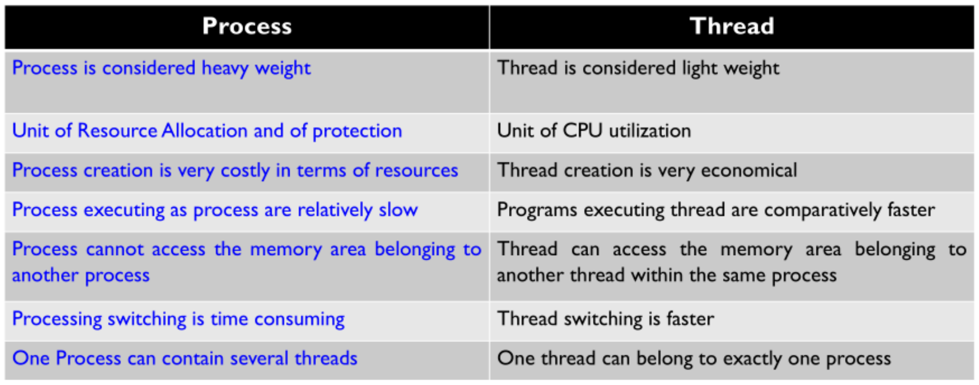
3.7 Thread Cancellation

- **Asynchronous cancellation** terminates the target thread immediately.
- **Feferred cacellation** allows the target thread to periodically check if it should be cancelled.

3.8 Thread Pools

- Create a number of threads in a pool where they await for works.
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread.
  - Allows the number of threads in the application(s) to be bound to the size of the pool.

3.9 Thread vs Process



Process	Thread
Process is considered heavy weight	Thread is considered light weight
Unit of Resource Allocation and of protection	Unit of CPU utilization
Process creation is very costly in terms of resources	Thread creation is very economical
Process executing as process are relatively slow	Programs executing thread are comparatively faster
Process cannot access the memory area belonging to another process	Thread can access the memory area belonging to another thread within the same process
Processing switching is time consuming	Thread switching is faster
One Process can contain several threads	One thread can belong to exactly one process

4 Process Synchronization

4.1 Background

- **Concurrent access** to shared data may result in **data inconsistency**
- Maintaining the data consistency requires mechanism to ensure the **orderly execution** of cooperating process.

4.2 Critical Section

```
do{
        entry section
        critical section
        exit section

        remainder section
} while(1);
```

- Mutual Exclusion: If process Pi is executing in it critical section, then **no other processes** can be executing in their critical sections.
- Progress: If no process is in its critical section or some processes that wish to enter their critical section,then **one** process will enter the critical section.
- Bounded Waiting: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section.

4.3 Peterson's Solution

- two processes solution.
- Two processes share two variables:

```
int turn;
bool falg[2];
```

- The variable **turn** indicates whose turn it is to enter the critical section.
- **flag[i]** implies that process Pi is ready.

```
do {
   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j) do-nothing;
        critical section
   flag[i] = FALSE;
   remainder section
} while (TRUE);
```

1. Mutual exclusion is preserved.
2. Progress requirement is satisfied.
3. Bounded-waiting requirement is met.

4.4 Synchronization Hardware

- Many systems provide hardware support for critical section code.
- **Uniprocessors**
  - Disable interrupts
  - Currently running code would execute without preemption.
  - Generally too efficient on multiprocessor systems.
- Modern machines provide special atomic hardware instructions.

4.5 Lock's Solution

```
do{
        acquire lock
        critical section
        release lock

        remainder section
} while(1);
```

- Peterson's Solution are not guaranteed to work on modern computer architecture.
- Any solution to the critical section problem requires a simple tool called **lock**.

5 Memory Management

5.1 Background

5.1.1 Basic Hardware

- Main memory and the registers that are built into each processing core are the only general-purpose storage that the CPU can access directly.
- Registers that are built into each CPU core are generally accessible within one circle of the CPU clock.
- Completing a memory access may take many cycles of the CPU clock. In such cases, the processor normally need to **stall**, since it does not have the data required to complete the instruction that it is executing.

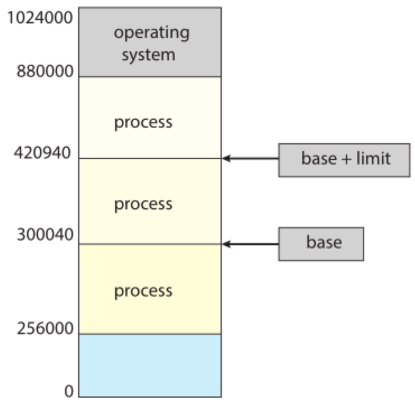


Figure 9.1 A base and a limit register define a logical address space.

For proper system operation, we must protect the operating system from access by user processes, as well as protect user processes from another

- Each process has a separate memory space, to separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses => Using two register.
- The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range.
- For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).
- Protection of memory space is accomplished by having the CPU hardware compare every address generated in user mode with the registers.
- The base and limit registers can be loaded only by the operating system, which use the a special privileged instruction.
- The operating system is given **unrestricted** access to both operating system memory and user memory.

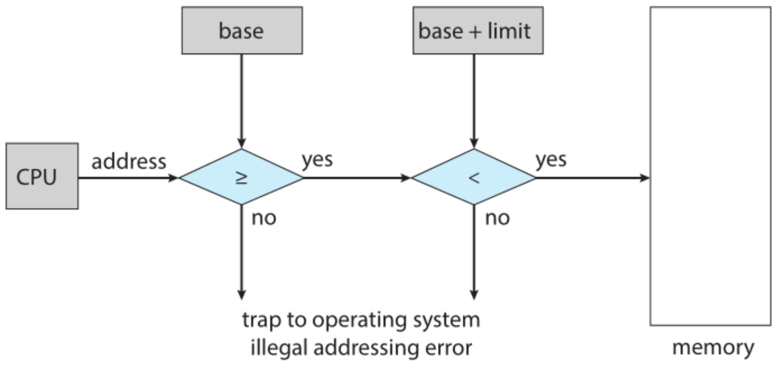


Figure 9.2 Hardware address protection with base and limit registers.

5.1.2 Address Binding

5.1.3 Logical and Physical Address Space

- An address generated by the CPU is commonly referred to as a **logical address**
- An address seen by the memory unit - the one loaded into the **memory-address register** of the memory - is called **physical address**.
- Logical address is sometimes referred as **virtual address**.
- All logical addresses is generated by a program called **logical address space**, the set of all physical addresses corresponding to these logical addresses is a **physical address space**.
- The run-time mapping from virtual to physical addresses is done by as hardware device called **memory-management unit (MMU)**.

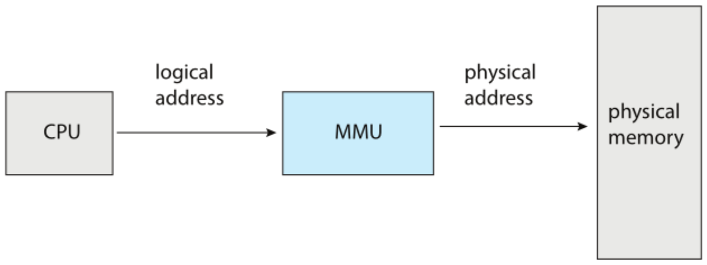


Figure 9.4 Memory management unit (MMU).

- We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a base value R). The user program generates only logical addresses and thinks that the process runs in memory locations from 0 to max.
- These logical addresses must be mapped to physical addresses before they are used.

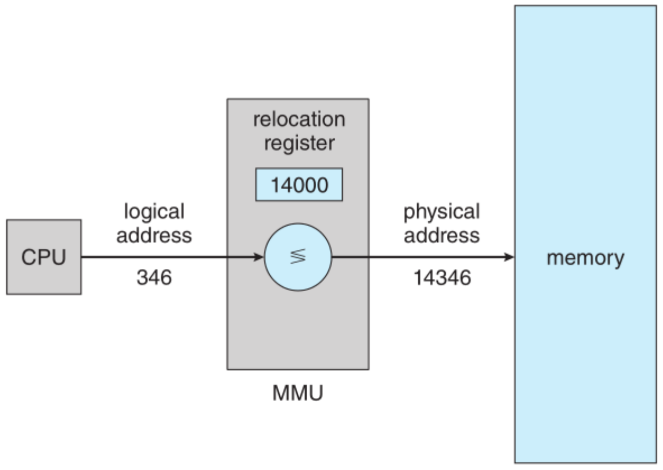


Figure 9.5 Dynamic relocation using a relocation register.

5.1.4 Dynamic Loading

The entire program and all data of a process to be in physical memory for the process to execute. The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use **dynamic loading**.

- A routine is not loaded until it is called. (advantage of dynamic loading)
- All routines are kept on disk in a relocatable load format.
- Dynamic loading does not require special support from operating system.

5.1.5 Dynamic Linking and Shared Libraries

- **Dynamically linked libraries (DLLs) are system libraries that are linked to user programs when the programs are run**
- **Static Linking:** system libraries are treated like any other object module and are combined by the loader into the binary program image.
- **Dynamic Linking:** linking is postponed until execution time.

5.2 Contiguous Memory Allocation

The memory is usually divided into two partitions: one for operating system and one for the user processeses. The operating system can be either low memory addresses or high memory addresses. For Linux and Window the operating systems are placed in high memory.

In **contiguous memory allocation**, each process is contained in a single section of memory that is contigous to the section containing the nest process.

5.2.1 Memory Protection

To prevent a process from accessing memory that it does not own, we use a relocation register and a limit register. The relocation register contains the smallest value of physical address; the limit register contains the range of the logical address.

Each logical addresss must fall within the range specified by the limit register.

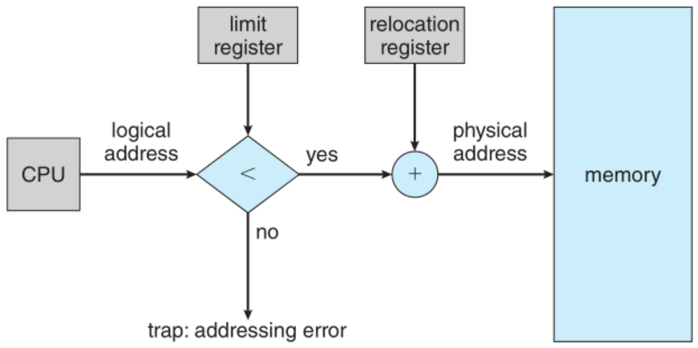


Figure 9.6 Hardware support for relocation and limit registers.

5.2.2 Memory Allocation

One of the simplest methods of allocating memory is to assign processes to variably sized partitions in memory, where each partition may contain exactly one process.

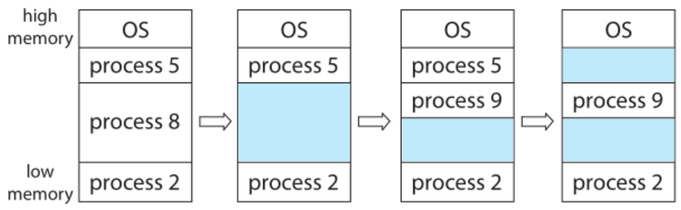


Figure 9.7 Variable partition.

If there is not suffcient memory to satisfy the demands of an arriving process, such process will be placed in to a wait queue. When memory is later released, the operating system checks the wait queue to determine if it will satisfy the memory demands of a waiting process.

The memory comprise a **set** of hole of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

The **first-fit**, **best-fit**, **worst-fit** strategies are the ones most commonly used to select a free hole from the set of available holes.

- **First-Fit:** Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- **Best-Fit:** Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- **Worst-Fit:** Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Both first-fit and best-fit are better than worst-fit in terms of decreasing time and storage untilization. Neither first-fit nor best-fit is clearly better than the other in terms of storage ultilization, but first-fit is generally faster.

5.2.3 Fragmentation

1. **External fragmentation:** exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. In the worst case, there is block of free(or wasted) memory between every two processes.
2. Both the **first-fit** and **best-fit** strategies for memory allocation suffer from external fragmentation.
3. **Internal fragmentation:** unused memory that is **internal** to a partition.

One solution for the external fragmentation is **compaction**. The goal is to **shuffle** the memory contents so as to place all free memory together in one large block. However it is only possible if relocation is dynamic and is done at the execution time.

Another possible solution is to permit the logical address space of processes to be noncontiguous, thus allowing a process to be allocated physical memory whereever such memory is available known as **paging**.

5.3 Paging

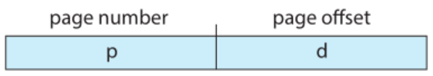
paging, a memory-management scheme that permits a process's physical address space to be non-contiguous. Paging avoids external fragmentation and the associated need for compaction.

5.3.1 Basic Method

The basic method for implement paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**.

When a process is to be executed,its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.

Every address generated by the CPU is divided into two parts: a **page number** (p) and a **page offset** (d)



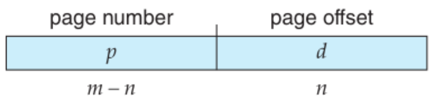
The page number is used as an index into a per-process **pafe table**. The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced. The base address of the frame is combined with the page offset to define the physical memory address.

How the MMU translate a logical address generated by the CPU to a physical address:

1. Extract the page number *p* and use it as an index into the page table.
2. Extract the corresponding frame number *f* from the page table.
3. Replace the page number *p* in the logical address with the frame number *f*.

The page size (the frame size) is defined by the hardware. The size of the page is a power of 2, typically varying between 4 Kb and 1 GB per page, depending on the computer architecture.

If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high order *m* - *n* bits is a page number, and the n low-order bits is the page offset.



When we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may face to some internal fragmentations.

If process size is indenpedent of page size, we expect internal fragmentation to average one-half page per process.

Pages is typically either 4 KB or 8 KB in size nowadays. On x86-64 systems, Windows 10 supports page sizes of 4 KB and 2 MB . Linux also supports two page sizes: a default page size (typically 4 KB)and an architecture-dependent larger page size called **huge pages**.

5.3.2 Hardware Support

page tables are per-process data structures, a pointer to the page table is stored with the other register values (like the instruction pointer) in the process conrol block of each process.

In the simplest case, the page table is implemented as a set of dedicated high-speed hardware registers, which makes the page-address translation very efficient. $\rightarrow$ increase context-switch time, as each one of these registers must be exchanged during a context switch.

The page table is kept in **main memory**, and a **page-table base register (PTBR)** points to the page table.

Changing page table requires only change this register, substaintially reducing context-switch time.

1. **Translation Loook-Aside Buffer (TLB)**

    - A special, small, fast-lookup hardware cache.
    - The TLB is associated and high-speed memory.
    - Each entry in the TLB contains two parts: a key(or tag) and a value.
    - The TLB must be kept small, typically between 32 and 1024 entries in size.

The TLB is used with page tables in the following way. The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU , the MMU first checks if its page number is present in the TLB . If the page number is found, its frame number is immediately available and is used to access memory.

IF the page table is not in the TLB (**TLB miss)**, address translation will proceeds with a page table.

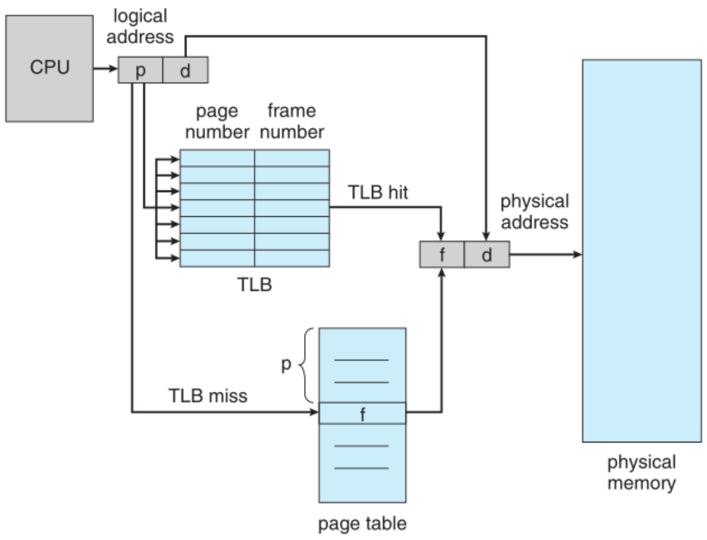


Figure 9.12 Paging hardware with TLB.

if the TLB is already full of entries, an existing entry must be selected for replacement.

The percentage of times that the page number of interest is found in the TLB is called the **hit ratio**

An 80-percent hit ratio, for example, means that we find the desired page number in the TLB 80 percent of the time. If it takes 10 nanoseconds to access memory, then a mapped-memory access takes 10 nanoseconds when the page number is in the TLB . If we fail to find the page number in the TLB then we must first access memory for the page table and frame number (10 nanoseconds) and then access the desired byte in memory (10 nanoseconds), for a total of 20 nanoseconds. The **effective memory access time** (EAT) is:

$$\begin{aligned} EAT &= 0.8 \times 10 + 0.2 \times 20 && (1) \\ &= 12 nanoseconds && (2) \end{aligned}$$

5.3.3 Protection

Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write is allowed.

At the same time that the physical address is being computed, the protection bis can be checked to verified that no writes are being made to a read-only page.

One additional bit is generally attached to each entry in the page table: a **valid-invalid** bit.

- When this bit is set to *valid*, the associated page is in the process's logical address space and is thus a legal (or valid) page.

- When the bit is set to *invalid*, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid-invalid bit.

Rerely does a process use all its address range. In fact, many processes use only a small fraction of the address space available to them.

Some system provide hardware, in the form of a **page-table-length register (PTLR)**, to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process

5.3.4 Shared Pages

- One copy of read-only (**reentrant**) code shared among processes (Ex: text editors, compilers, window systems)
- Similar to multiple threads sharing the same process space.
- Useful for inter-process communication is sharing of read-only pages is allowed.

5.4 Structure of the Page Table

5.4.1 Hierarchical Paging

- Most modern computer systems support a large logical address space ($2^{32} to 2^{64}$). $\rightarrow$ the page table itself becomes excessively large.
- One way to solve this problem is use a two-level paging algorithm, in which the page table itself is also paged.
- For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate.

5.4.2 Hashed Page Tables

- One approach for handling address spaces larger than 32 bits is to use a **hashed page table**, with the hash value being the virtual table number.
- Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions).
- Each element consists of three fields:
  - The virtual page number.
  - The value of the mapped page frame.
  - A pointer to the next element in the linked list
- The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

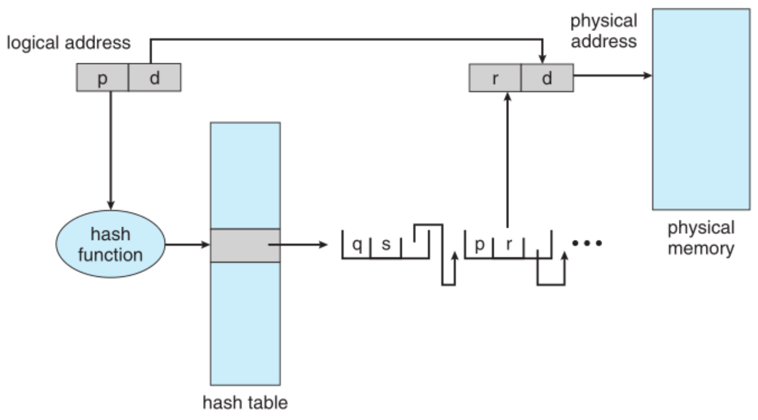


Figure 9.17 Hashed page table.

- A variation of this scheme that is useful for 64-bit address spaces has been proposed. This variation uses clustered page tables, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page.

5.5 Swapping

- Process instructions and the data they operate on must be in memory to be executed. However, a process, or a portion of a process, can be **swapped** temporarily out of memory to a **backing store** and then brought back into memory for continue execution.

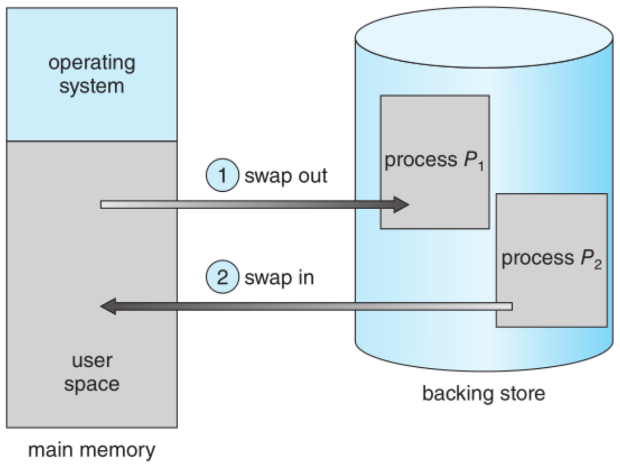


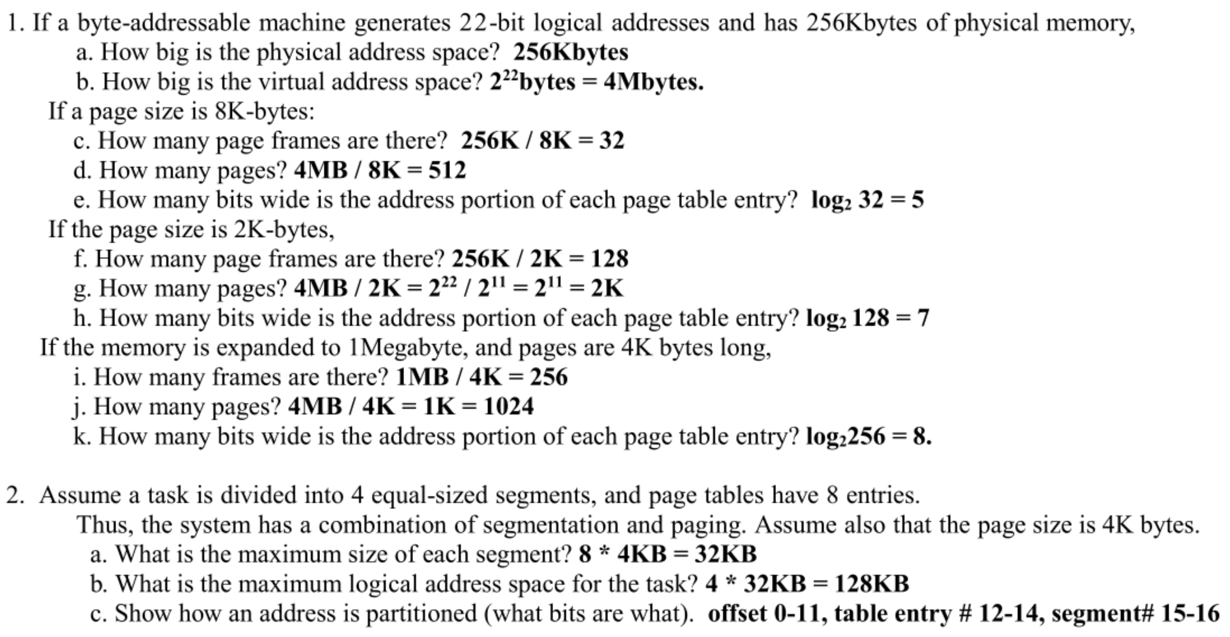
Figure 9.19 Standard swapping of two processes using a disk as a backing store.

- Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system.

- **Roll in, roll out:** lower-priority process is swapped out so higher-priority process can be loaded and executed.
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk.

5.6 Segmentation

- Memory-management scheme that supports user view of memory.
- A program is a collection of segments; A segment is a logical unit such as: main program, procedure, function, method, object, local variables, global variables, stack, etc.



1. If a byte-addressable machine generates 22-bit logical addresses and has 256Kbytes of physical memory,
   a. How big is the physical address space? **256Kbytes**
   b. How big is the virtual address space? **2^{22}bytes = 4Mbytes.**
   If a page size is 8K-bytes:
   c. How many page frames are there? **256K / 8K = 32**
   d. How many pages? **4MB / 8K = 512**
   e. How many bits wide is the address portion of each page table entry? **$\log_2 32 = 5$**
   If the page size is 2K-bytes,
   f. How many page frames are there? **256K / 2K = 128**
   g. How many pages? **4MB / 2K = $2^{22} / 2^{11} = 2^{11}$ = 2K**
   h. How many bits wide is the address portion of each page table entry? **$\log_2 128 = 7$**
   If the memory is expanded to 1Megabyte, and pages are 4K bytes long,
   i. How many frames are there? **1MB / 4K = 256**
   j. How many pages? **4MB / 4K = 1K = 1024**
   k. How many bits wide is the address portion of each page table entry? **$\log_2 256 = 8$.**

2. Assume a task is divided into 4 equal-sized segments, and page tables have 8 entries.
   Thus, the system has a combination of segmentation and paging. Assume also that the page size is 4K bytes.
   a. What is the maximum size of each segment? **8 * 4KB = 32KB**
   b. What is the maximum logical address space for the task? **4 * 32KB = 128KB**
   c. Show how an address is partitioned (what bits are what). **offset 0-11, table entry # 12-14, segment# 15-16**

For each of the following convert the virtual address into a physical address (if possible) and write down the value of the memory location corresponding to the address. If it is not possible to do so, explain why.

a.) 0x2AA4 (1 0 1 0 1 0. 1 0 1 0 0 1 0 0 in binary).
Segment entry# = 1, Table entry#: $01010_2 = 10_{10}$, Offset: $10100100 = A4_{16}$. Page table 2 is present. Page Table 2 entry 10 is located at frame 0x4. So the physical address is 0x4A4, which has value 0x30

b.) 0x6CA4 (1 1 0 1 1 0 0 1 0 1 0 0 1 0 0 in binary).
Segment entry# = $11_2 = 3_{10}$, Table entry# = $01100_2 = 12_{10}$, Offset = $10100100 = A4_{16}$. The page table for segment table entry 3 is not present in RAM.

c.) 0x59A4 (1 0 1 1 0 0 1 1 0 1 0 0 1 0 0 in binary).
Segment entry# = $10_2 = 2_{10}$, Table entry# = $11001_2 = 25_{10}$, Offset = $10100100 = A4_{16}$. Page table 0 is present. Page Table 0 entry 25 is not present in RAM.

d.) 0xEFA4 (1 1 1 0 1 1 1 1 1 0 1 0 0 1 0 0 in binary).
Segment entry # = $111_2 = 7_{10}$, Table entry # = $01111_2 = 15_{10}$, Offset = $10100100 = A4_{16}$. Page table 5 is present. Page Table 5 entry 15 is located at frame 0x31. So the physical address is 0x31A4, which has the value 0x74.

4. Page Replacement Policy: Assume a page address stream of 2 3 1 5 3 2 4 1 3 2 1 4 for a system with three page frames allocated to the process. Fill in the following table based on the Clock (or any other page replacement policy specified on the final) algorithm. The first three are already filled in.

Page addressed	**2**	**3**	**1**	**5**	**3**	**2**	**4**	**1**	**3**	**2**	**1**	**4**
Entry 0	**2**			**5**				**1**				
Entry 1		**3**					**4**			**2**		
Entry 2			**1**			**2**			**3**			**4**
Fault? (Y or N)	**Y**	**Y**	**Y**	**Y**	**N**	**Y**	**Y**	**Y**	**Y**	**Y**	**N**	**Y**

Good Luck :) Myself