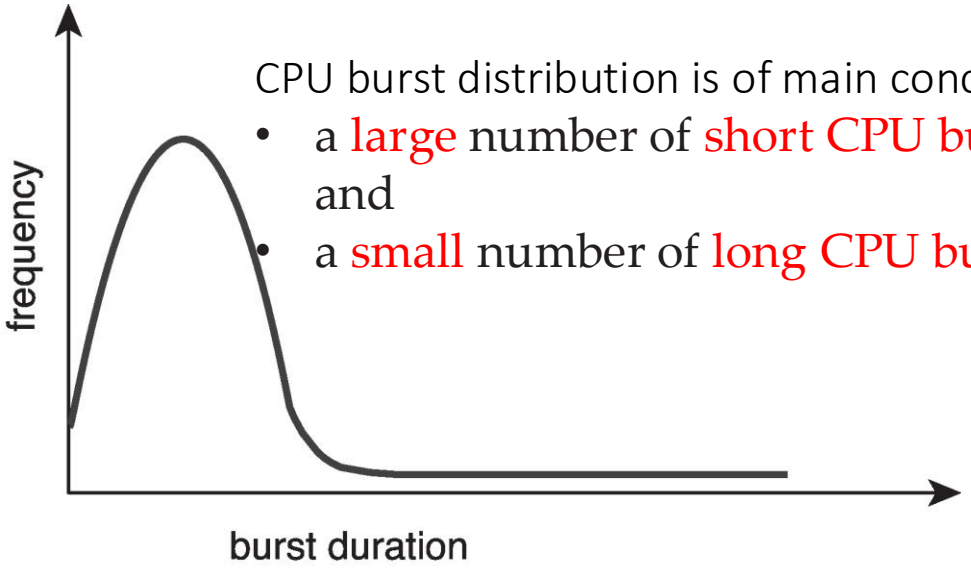
Chapter 5: CPU Scheduling

Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples

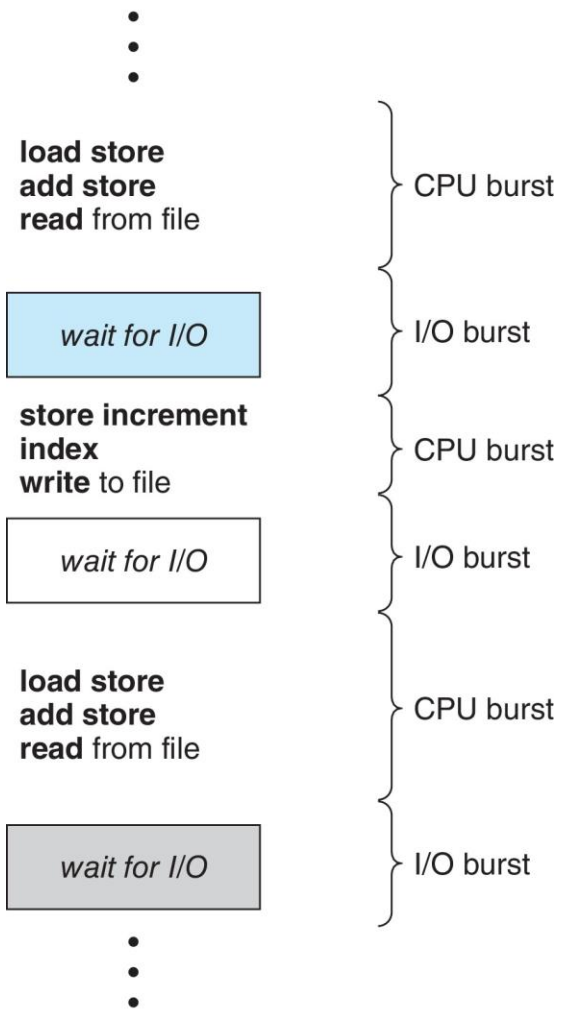
Basic Concepts

- Purpose of multiprogramming: maximum CPU utilization
- CPU–I/O Burst Cycle
  - Process execution consists of a cycle of CPU execution and I/O wait
  - CPU burst followed by I/O burst

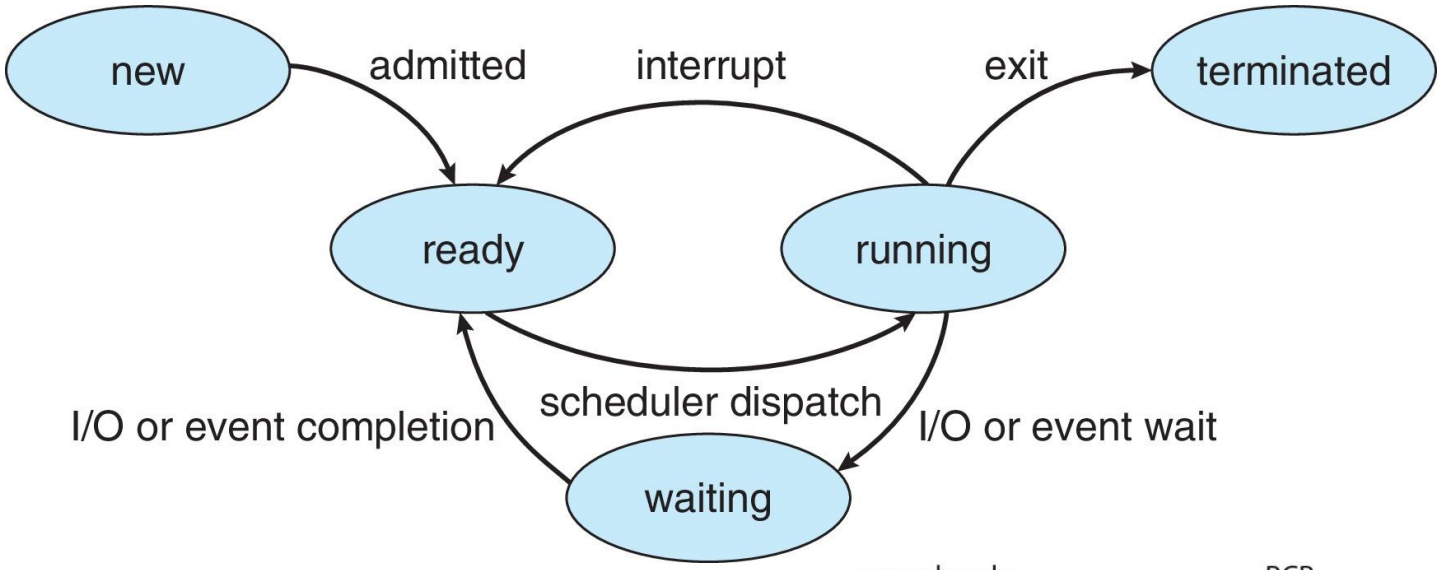


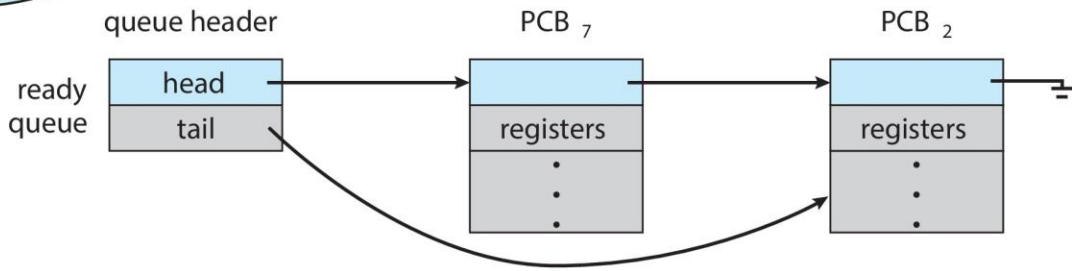
CPU burst distribution is of main concern:

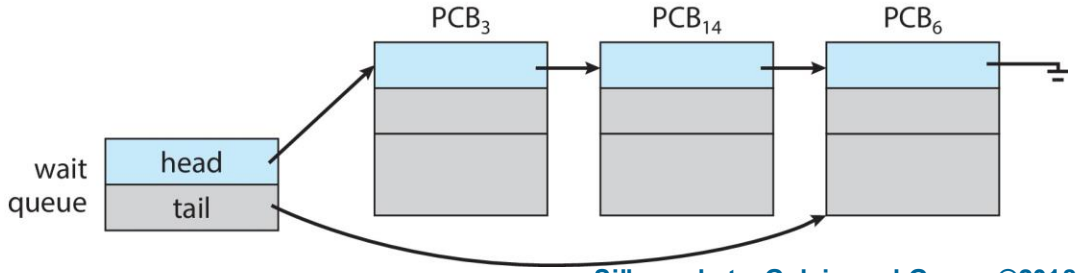
- a large number of short CPU bursts and
- a small number of long CPU bursts.



Process State Transition







CPU Scheduler

- The CPU scheduler (CPU 调度程序) selects a process from the processes in ready queue, and allocates the CPU to it
  - Ready queue may be ordered in various ways
- CPU scheduling decisions may take place when a process
  1. switches from running to waiting state (non-preemptive 自愿离开CPU)
     - Example: the process does an I/O system call.
  2. switches from running to ready state (preemptive 强占)
     - Example: there is a clock interrupt.
  3. switches from waiting to ready (preemptive)
     - Example: there is a hard disk controller interrupt because the I/O is finished.
  4. terminates (non-preemptive 自愿离开CPU)

CPU Scheduler

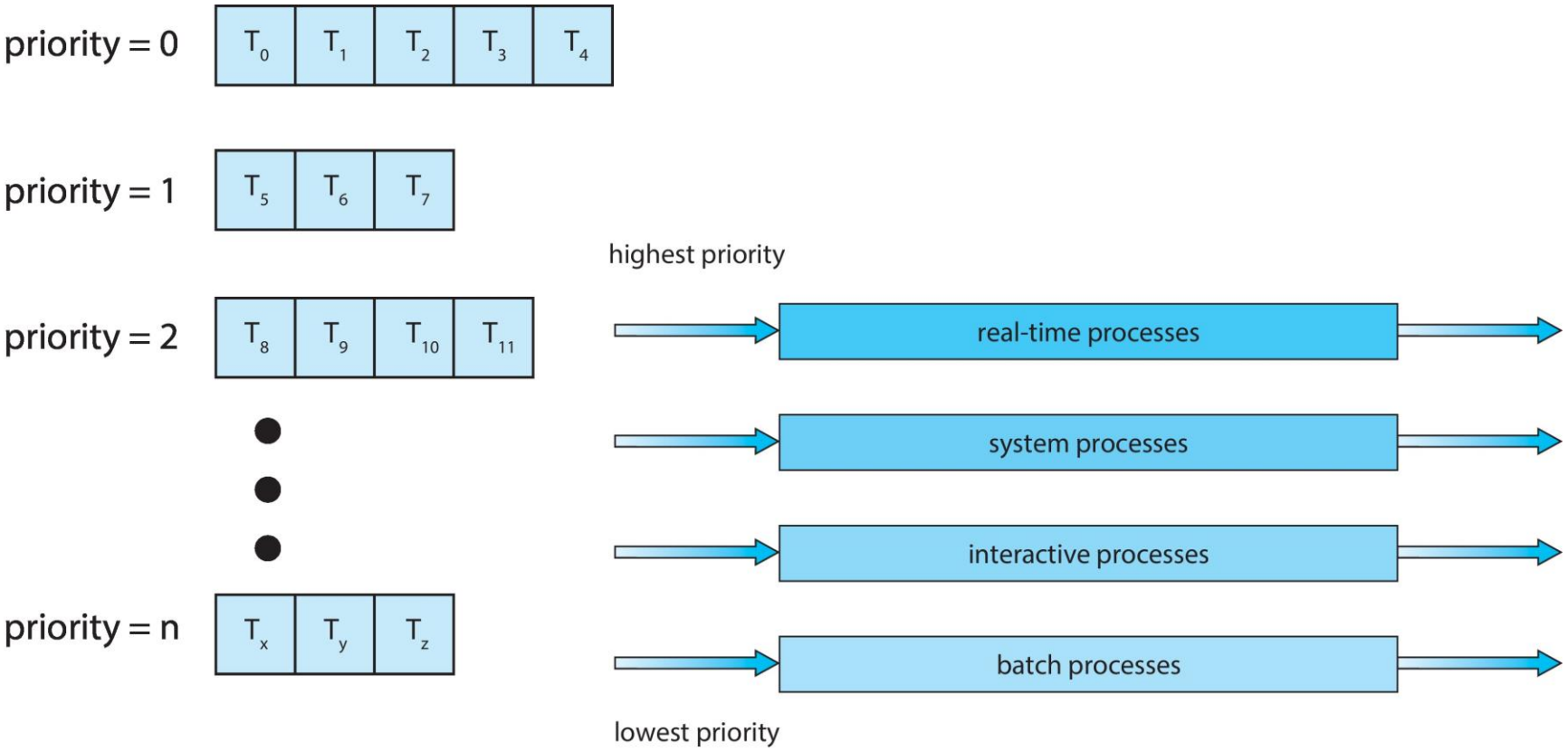
- Scheduling under 1 and 4 is non-preemptive (非强占的, decided by the process itself)
- All other scheduling is pre-emptive (强占的, decided by the hardware and kernel)
- Preemptive scheduling can result in race conditions (will introduced in chapter 6) when data are shared among several processes
- Some considerations in pre-emptive scheduling
  1. Access to shared data
  2. Preemption issue while CPU is in kernel mode
  3. How to handle interrupts during crucial OS activities

Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
  - foreground (interactive 交互processes)
  - background (batch 批处理 processes)
- Process permanently in a given queue (stay in that queue)
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling
    - Each queue has a given priority
      - High priority queue is served before low priority queue
      - Possibility of starvation
  - Time slice
    - Each queue gets a certain amount of CPU time

Multilevel Queue

- With priority scheduling, for each priority, there is a separate queue
- Schedule the process in the highest-priority queue!



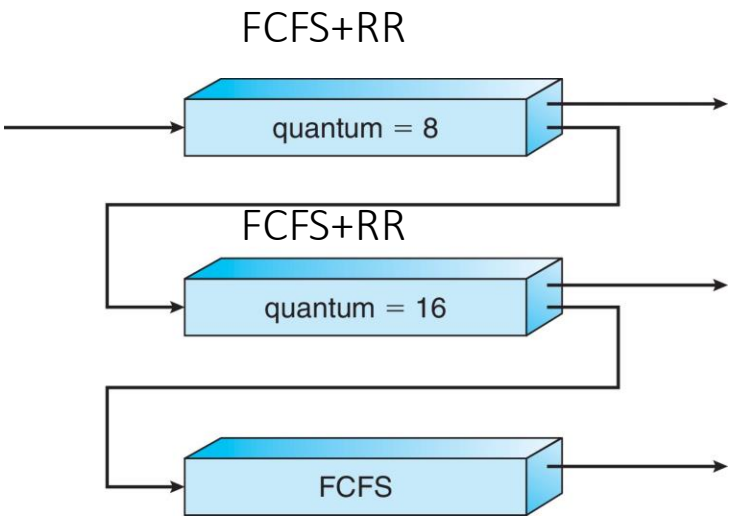
Prioritization based upon **process type**

Multilevel Feedback Queue

- A process can move between the various queues;
  - aging can be considered in this way (prevent starvation)
  - advantage: prevent starvation
- The multilevel feedback queue scheduler
  - the most general CPU scheduling algorithm
  - defined by the following parameters:
    1. number of queues
    2. scheduling algorithms for each queue
    3. Policies on moving process between queues
       1. when to upgrade a process
       2. when to demote (降级) a process
       3. which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  1. Q_0 – RR with time quantum 8 milliseconds
  2. Q_1 – RR with time quantum 16 milliseconds
  3. Q_2 – FCFS



- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2 where it runs until completion but with a low priority

Multiple-Processor Scheduling – Load Balancing

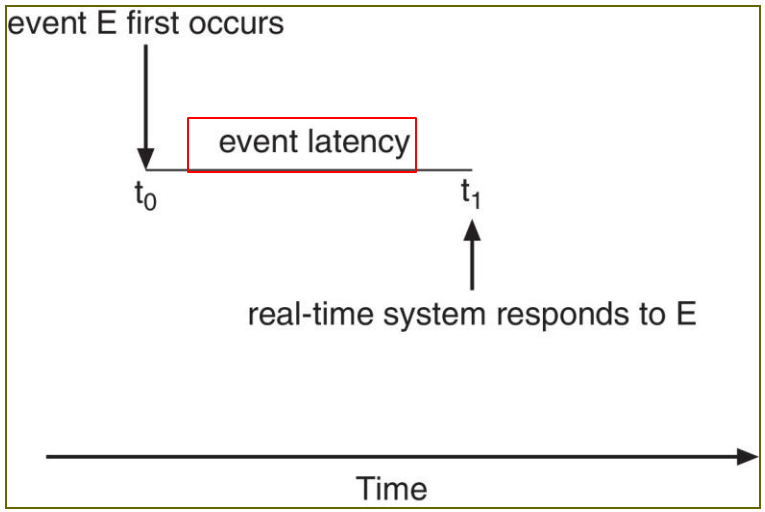
- If **Symmetric multiprocessing** needs to keep all CPUs loaded for efficiency,
- Load balancing attempts to keep workload evenly distributed
  - Push migration – periodic task checks load on each processor, and pushes tasks from overloaded CPU to other less loaded CPUs
  - Pull migration – idle CPUs pulls waiting tasks from busy CPU
- Push and pull migration need not be mutually exclusive
  - They are **often implemented in parallel** on load-balancing systems.

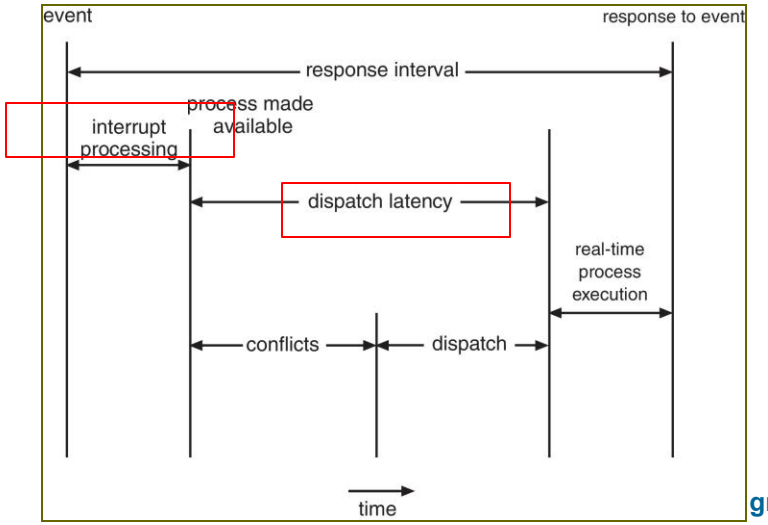
Multiple-Processor Scheduling – Processor Affinity

- Processor affinity
  - When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread, i.e., a thread has affinity for a processor
- Load balancing may affect processor affinity
  - a thread may be moved from one processor to another to balance loads,
  - that thread loses the contents of what it had in the cache of the processor it was moved off
- Soft affinity – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- Hard affinity – allows a process to specify a set of processors it may run on.
  - The kernel then never moves the process to other CPUs, even if the current CPUs have high loads.

Real-Time CPU Scheduling

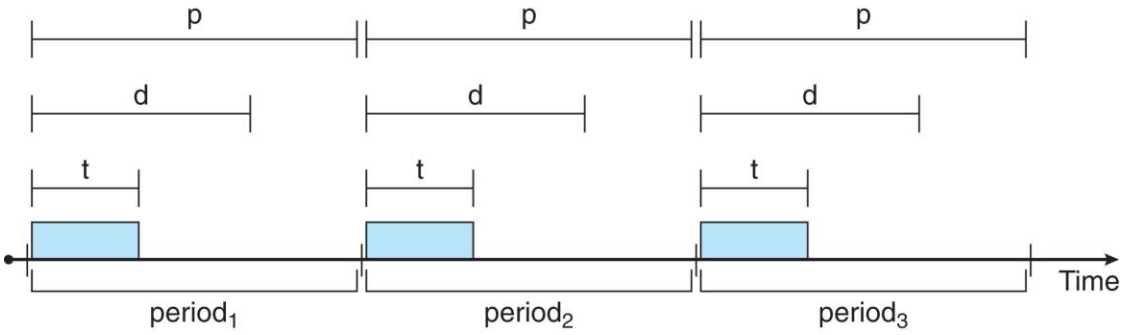
- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of kernel interrupt service routine (ISR) that services interrupt
  2. Dispatch latency(调度延迟) – time for scheduler to take current process off CPU and switch to another





Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
  - For hard real-time, must also provide ability to meet deadlines
- Processes have new characteristics: periodically require CPU at constant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \leq t \leq d \leq p$
  - Rate of periodic task is $1/p$



Rate Monotonic Scheduling

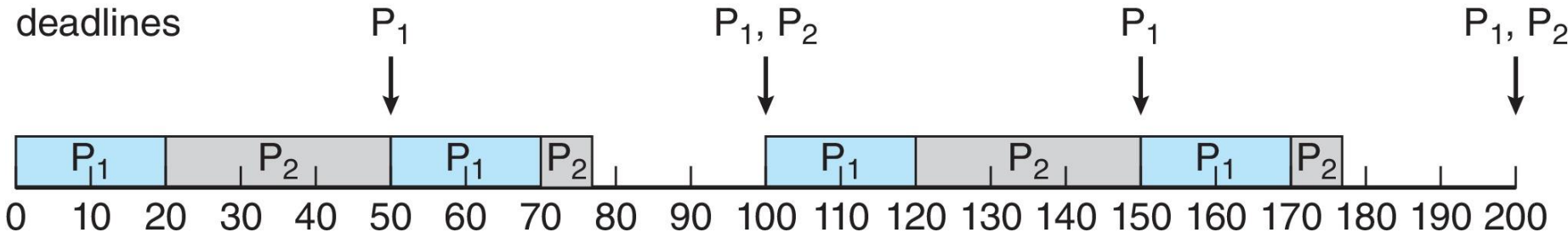
- A priority is assigned based on the inverse of its period

  Shorter periods = higher priority

  Longer periods = lower priority

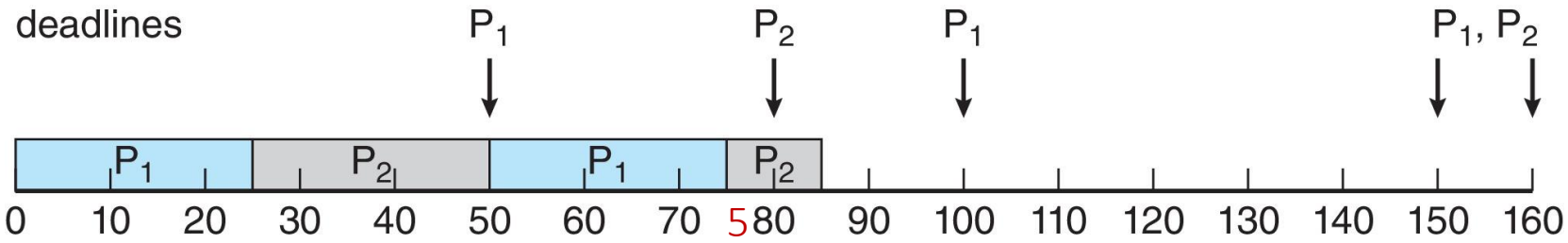
- In the following example, P_1 is assigned a higher priority than P_2.
  - P_1 needs to run for 20 ms every **50** ms. t = 20, d = p = 50
  - P_2 needs to run for 35 ms every **100** ms. t = 35, d = p = 100

  Assume deadline d = p



Missed Deadlines with Rate Monotonic Scheduling

- Example:
  - P_1 needs to run for 25 ms every 50 ms. t = 25, d = p = 50
  - P_2 needs to run for 35 ms every 80 ms. t = 35, d = p = 80

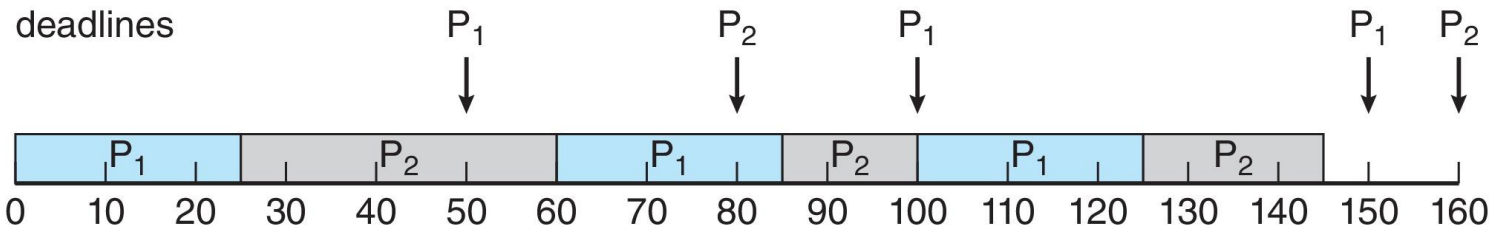


P2:25 P2:needs 10

- Process P_2 misses its deadline at time 80 ms.
- Observation: if P_2 is allowed to run from 25 to 60 and P_1 then runs from 60 to 85 then both processes can meet their deadline.
- So the problem is not a lack of CPU time, the problem is that rate monotonic scheduling is not a very good algorithm.

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - the earlier the deadline, the higher the priority;
  - the later the deadline, the lower the priority.



- Example:
  - P_1 needs to run for 25 ms every 50 ms.
  - P_2 needs to run for 35 ms every 80 ms.
- This is the scheduling algorithm many students use when they have multiple deadlines for different homework assignments!

Chapter 6 & 7: Process Synchronization

Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
  - Concurrent access to shared data may result in data inconsistency
- An example
  - The variable counter in an implementation of producer-consumer

```
item  next_produced;
while (true) {
    /* produce an item in next_produced */
    while (counter == BUFFER_SIZE) //buffer is full
        ;/* do nothing, such as create data */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
item  next_consumed;
while (true) {
    while (counter == 0) //buffer is empty
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;  //total # of items in the buffer
    /* consume the item in next_consumed */
}
```

Race Condition

- counter++ could be implemented as
  - register1 = counter
  - register1 = register1 + 1
  - counter = register1
- counter-- could be implemented as
  - register2 = counter
  - register2 = register2 - 1
  - counter = register2
- Consider this execution interleaving with "counter = 5" initially:
  - S0: producer execute register1 = counter {register1 = 5}
  - S1: producer execute register1 = register1 + 1 {register1 = 6}
  - S2: consumer execute register2 = counter {register2 = 5}
  - S3: consumer execute register2 = register2 – 1 {register2 = 4}
  - S4: producer execute counter = register1 {counter = 6 }
  - S5: consumer execute counter = register2 {counter = 4}

Race condition:

- several processes access and manipulate the same data concurrently
- outcome depends on which order each access takes place.

Critical Section Problem

- Consider system of *n* processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Several processes may be changing common variables, updating table, writing file, etc.
  - critical section
    - The segment of code in a process that modifies these shared variables, tables, files
- When one process is in critical section, other process should not enter their critical sections for these shared data .

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Critical Section

- General structure of process P_i

```
do {
    entry section
        critical section
    exit section
    remainder section
} while (true);
```

Ask for permission

Protect this section

Do sth in order to allow other processes to enter critical section

Critical Section Problem

- Critical section problem is to design a protocol（协议）that the processes can use to cooperate
  - Process with critical section should follow the following steps
    1. execute entry section to ask for permission
    2. then execute critical section,
    3. execute exit section to allow other process to enter critical section
    4. then execute remainder section

Solution to Critical-Section Problem

- A solution to the critical-section problem must satisfy three requirements:
  1. Mutual Exclusion
     - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
  2. Progress
     - If no process is executing in its critical section and if other processes want to enter critical section, one of them must be selected. They cannot be postponed indefinitely
  3. Bounded Waiting
     - If a process has made a request to enter its critical section, then, before that request is granted, there is a bound to the times that others can enter critical section

Solution to Critical-Section Problem

- Assumptions in the solution
  - each process executes at a nonzero speed
  - no concerning relative speed of the *n* processes
  - the **load** and **store** machine-language instructions are atomic (that is, cannot be interrupted)

Example: Solution to Critical-Section Problem

Example: two students want to eat from the same plate of food using one spoon.

1. **Mutual Exclusion** - If one student is eating using the spoon, then the other student must wait for the spoon to become available: the two students cannot both use the spoon at the same time! Otherwise the two students will end up fighting and breaking the plate of food (race condition).
2. **Progress** - If one student is not hungry and the other student wants to get the spoon to eat, then the other student must be able to take the spoon and eat: the student who is not hungry is not allowed to prevent the other student from taking the spoon and eating. Otherwise the other student will wait for ever and die from hunger.
3. **Bounded Waiting** - If one student is eating and the other student is waiting for the spoon, the first student is not allowed to put down the spoon on the table and immediately take it again to eat again: the student who puts down the spoon must allow the other student to take the spoon so that the other student can eat too. In other words, the two students must be nice with each other and not be selfish. Otherwise the other student will wait for ever and die from hunger.

Critical Section: Solution 1

turn's value is initialized to be either *i* or *j*

P_i	P_j

do {
while (turn == j); //entry section
critical section //turn is i
turn = j; //exit section
remainder section
} while (true);
	do {
	while (turn == i); //entry section
	critical section // turn is j
	turn =i; //exit section
	remainder section
	} while (true);

Mutual Exclusion: Yes

Progress: No

Bounded waiting: Yes

Detailed Explanation on Example 1

1. Mutual Exclusion: Yes - If process P_i is inside the critical section then P_j cannot enter the critical section because of the **`while`** loop in the entry section that forces P_j to wait for P_i to exit the critical section.
    1. And vice-versa: if P_j is inside the critical section then P_i must wait in the entry section until P_j exits the critical section.
    2. What happens if both processes try to enter the critical section at the exact same time? The shared **`turn`** variable can only store **`i`** or **`j`**, but not both, so the test of the **`while`** loop will only be false for one of the two processes, not for both, and only one process will be able to enter the critical section, the other process will have to wait in the entry section.
        1. Which process enters the critical section first is then determined by the initial value of the **`turn`** variable.

Detailed Explanation on Example 1 (Continue)

2. Progress: No – If the **`turn`** variable contains the value **`j`** (for example) and P_i wants to enter the critical section and P_j does not want to enter the critical section (assuming there is no **`do{}while`** loop around P_j's code and P_j is busy doing something else) then P_i will wait for ever, even though P_j is not inside the critical section.
   1. P_i can only enter the critical section after P_j has been in the critical section, and P_j can only enter the critical section after P_i has been in the critical section.

      $P_i \rightarrow P_j \rightarrow P_i \rightarrow P_j \rightarrow P_i \rightarrow P_j \rightarrow P_i \rightarrow \cdots$
   2. If P_j does not want to enter the critical section again then P_i is going to get stuck and wait for ever in the entry section for P_j to exit the critical section, which P_j is never going to do.

      $P_i \rightarrow P_j \rightarrow P_i \rightarrow P_i$ again and gets stuck for ever.

Detailed Explanation on Example 1 (Continue)

3. Bounded Waiting: Yes - There is a bound of 1 time on the number of times that P_j (for example) is allowed to enter the critical section after P_i has made a request to enter its critical section and before that request is granted, because when P_j exits the critical section it changes the **`turn`** variable to be **`i`**, which then immediately allows P_i to enter the critical section.
    1. Even if we assume that P_j is very fast and P_i is very slow (for example), it is not possible for P_j to beat P_i to the race and exit the critical section and immediately re-enter it again before P_i is allowed to take its turn inside the critical section.
    2. In other words, it is not possible for P_j to prevent P_i from going into the critical section by going in and out of the critical section as fast as possible.
    3. So P_i is guaranteed to be able to enter the critical section at some point in the future (right after P_j is finished) and will not have to wait for ever, even if P_j wants to re-enter the critical section all the time.

Peterson's Solution

- It is a classic software-based solution to the critical-section problem
  - Good algorithmic description of solving the problem
- Solution for two processes by using two variables:
  - int turn; // indicates whose turn it is to enter the critical section.
  - Boolean flag[2] // indicate if a process is ready to enter the critical section.
    - flag[i] = *true* implies that process P_i is ready!
    - It is initialized to FALSE.

Algorithm for Process P_i & P_j

P_i

```
do {
        flag[i] = true; //ready
        turn = j; //allow Pj to enter
        while (flag[j] && turn = = j);
        critical section
        flag[i] = false; //exit
        remainder section
} while (true);
```

Ask for entry permission

Mutual Exclusion: Yes

Progress: Yes

Bounded waiting: Yes

P_j

```
do {
        flag[j] = true; //ready
        turn = i; //allow Pi to enter
        while (flag[i] && turn = = i);
        critical section
        flag[j] = false; //exit
        remainder section
} while (true);
```

Ask for entry permission

Synchronization Hardware

- Software-based solutions (such as Peterson's) are not guaranteed to work on modern computer architecture
- Many systems provide **hardware support** for implementing the critical section code.
  - **Uniprocessors** – could disable interrupts
    - Currently running code would execute without preemption
      - **too inefficient**
  - Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible, the atomic hardware instruction will do the following work
      1. Test memory *word* and set value
      2. Swap contents of two memory *words*
    - E.g., test_and_set instruction in the next page

Atomic test_and_set Instruction

Definition:

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = TRUE;
        return rv:
}
```

1. The C code here is just to explain how the hardware instruction works.
2. This instruction is executed atomically by CPU as a single hardware instruction.
3. In practice, *target* is a pointer to the lock itself, shared by all the processes that want to acquire the lock.
   - If target if *FALSE*, the return value of *rv* is *FALSE*, means lock is *FALSE* (available), target's new value is *TRUE*
   - If *target* is true, the return value of *rv* is *TRUE*, means lock is *TRUE* (locked)

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
        critical section
    release lock
        remainder section
}  while (TRUE);
```

entry section

exit section

- Use the idea of locking
  - Protecting critical regions via locks
- A process that wants to enter the critical section must first get the lock.
- If the lock is already acquired by another process, the process will wait until the lock becomes available.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution using test_and_set:

```
do {
        while (test_and_set(&lock))
              ; /* do nothing */

        /* critical section */

        lock = FALSE;

        /* remainder section */

} while (true);
```

```
boolean test_and_set
(boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

```
do {
    acquire lock

    critical section

    release lock

    remainder section

} while (TRUE);
```

- When lock = true, keep while looping.
- When lock = FALSE, process can enter the critical section
- And set lock = TRUE, block other processes to enter.
- After finish the critical section, reset lock = FALSE, to allow other processes to enter the critical section.

Mutual Exclusion: Yes

Progress: Yes

Bounded waiting: No

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

   /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;

   /* remainder section */
} while (true);
```

P_i

```
do {
      acquire lock
      critical section
      release lock
      remainder section
} while (TRUE);
```

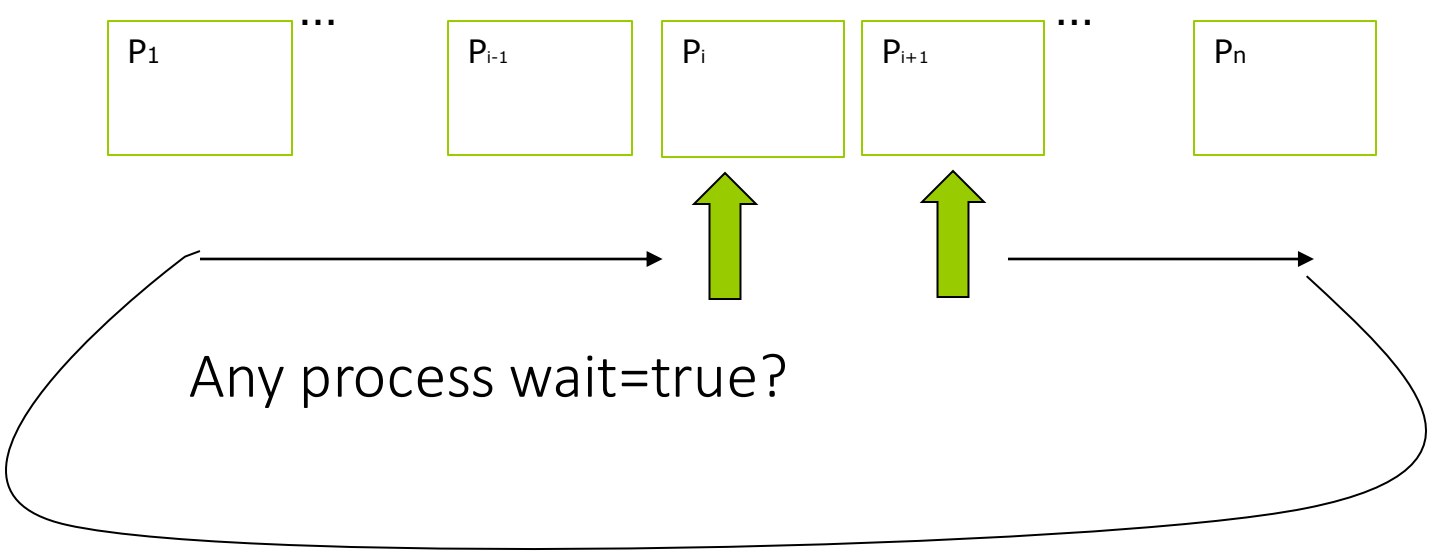
Common data structures:

boolean waiting[n];

boolean lock;

These data structures are initialized to **false**.

Bounded-waiting Mutual Exclusion with test_and_set (Continue)



Bounded-waiting Mutual Exclusion with test_and_set (Continue)

P_i

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

   /* critical section
*/

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;

   /* remainder section */
} while (true);
```

P_k

```
do {
   waiting[k] = true;
   key = true;
   while (waiting[k] && key)
      key = test_and_set(&lock);
   waiting[k] = false;

   /* critical section */

   j = (k + 1) % n;
   while ((j != k) && !waiting[j])
      j = (j + 1) % n;
   if (j == k)
      lock = false;
   else
      waiting[j] = false;

   /* remainder section */
} while (true);
```

Mutual exclusion: Yes Progress: Yes Bounded-waiting: Yes

Explanations on Three Reequipments for Slide 24

Mutual exclusion: Yes

```
waiting[i] = true;
key = true;
while (waiting[i] && key)
   key = test_and_set(&lock);
waiting[i] = false;
```

1. Process P_i can enter its critical section only if either waiting[*i*] == false or key == false
   - If waiting[*i*] is false, that means, another process finished critical section and give the turn to process P_i
   - If key is false, that means, lock was false before run test_and_set.
2. If another process (e.g., P_k) in waiting to enter, it cannot enter. Because waiting[*k*] is true, and lock is true.

Bounded-waiting Mutual Exclusion with test_and_set

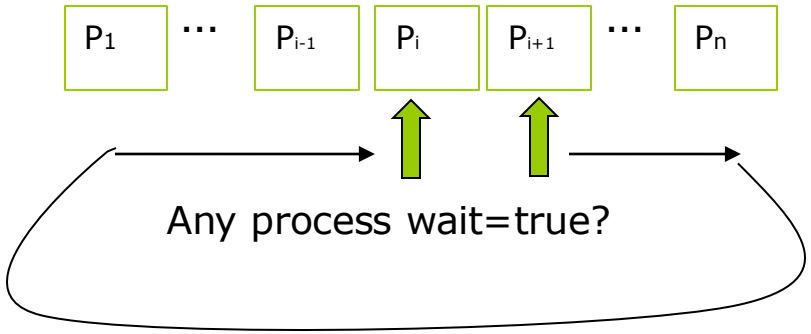
Progress: Yes

If P_i wants to enter the critical section, while other process do not want to enter, lock is false. So the test_and_set will return false.

Bounded waiting: Yes

If process P_i leaves the critical section, and another process (e.g. P_k) is waiting, the exit section in P_i will set waiting[k] to the false

```
j = (i + 1) % n;
while ((j != i) && !waiting[j])
  j = (j + 1) % n;
if (j == i)
    lock = false;
else
    waiting[j] = false;
```



Semaphore（信号灯）

- Semaphore is a synchronization tool
  - more sophisticated than mutex locks
- Semaphore *S:* an integer variable
  - S can only be accessed via two atomic operations
    - `wait()` and `signal()`
      - Originally called `P()` and `V()`

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

acquire lock

```
signal(S) {
    S++;
}
```

release lock

Semaphore（信号灯）

P_i

```
do {
   wait(S);
   /* critical section */
   signal(S)
   /* remainder section */
} while (true);
```

Pj

```
do {
   wait(S);
   /* critical section */
   signal(S)
   /* remainder section */
} while (true);
```

```
wait(S) {
     while (S <= 0)
        ; // busy wait
     S--;
}
```

```
signal(S) {
     S++;
}
```

release lock

sem_wait and sem_post

```
#include <semaphore.h>
sem_t *sem_mutex;
```

Normally used among different processes

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

```
sem_wait(sem_mutex);
```

```
sem_post(sem_mutex);
```

```
do {
    acquire lock
        critical section
    release lock
      remainder section
} while (true);
```

Semaphore Usage

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1
  - Same as a mutex lock
- Semaphore can be used to solve various synchronization problems

An example: Consider two processes P_1 and P_2 that require T_1 to happen before T_2. Solution:

```
semaphore synch;
synch = 0
```

```
P1:
    T1;
    signal(synch);
```

```
P2:
    wait(synch);
    T2;
```

Semaphore Implementation

- Advantage
  - implementation code is short
- Disadvantage :
  - Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
    - S--, S++ in wait() and signal()
    - Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
    - The solution now have busy waiting in critical section implementation
  - Both mutex lock and semaphore suffer from busy waiting.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
  - Each semaphore has:
    - an integer value
    - A queue of processes
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue

```
typedef struct{
      int value;
      struct process *list;
} semaphore;
```

Implementation with no Busy waiting (Cont.)

- Again, C code is just for explanation:

Busy waiting

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}

signal(S) {
    S++;
}
```

S: initial value 1

No busy waiting

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
       //add this process to S->list;
       block();//put in a waiting queue
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
     //remove a process P from S->list;
       wakeup(P);//put in a ready queue
    }
}
```

S->value: initial value 1

Classical Problems of Synchronization

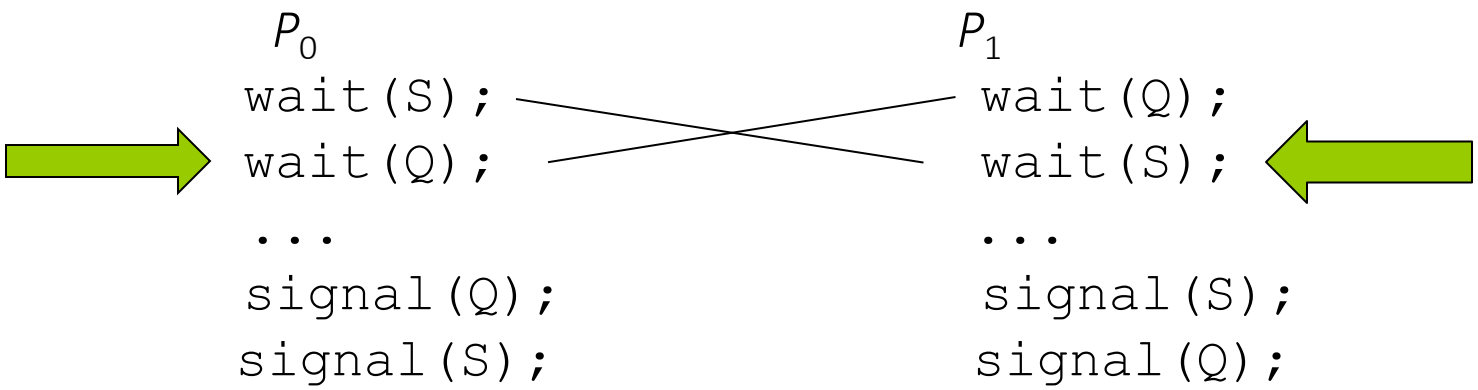
- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers-Writers Problem
  - Dining-Philosophers Problem

Problems with Semaphores

- Deadlock and starvation are possible.
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- It is the responsibility of the application programmer to use semaphores correctly.

Deadlock and Starvation

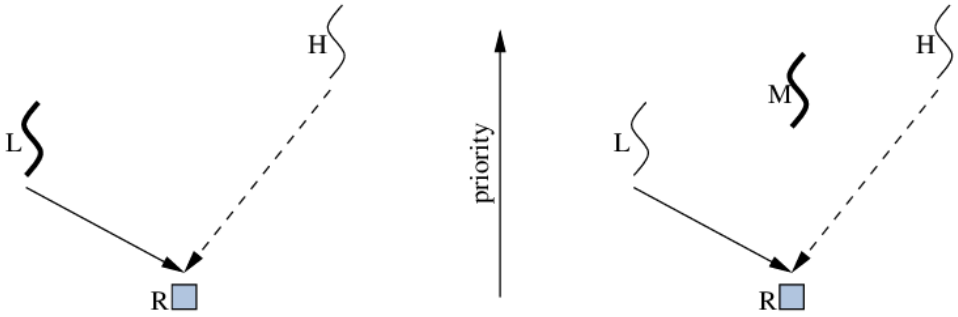
- Let S and Q be two semaphores initialized to 1



- Can cause starvation and priority inversion
  - Starvation – indefinite blocking
    - A process may never be removed from the semaphore queue in which it is suspended)
  - Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Deadlock and Starvation

- priority inversion example:
  - Assume three processes
    - *L*, *M*, and *H with* the priority order: *L* < *M* < *H*.
  - Assume that
    - process *H* requires resource *R*, which is currently hold by process *L*.
    - process *H* would wait for *L* to finish using resource *R*.
  - However, process *M* becomes runnable, and preempts process *L*.
  - Consequence:
    - process *M* (with middle priority) has affected how long process *H* must wait for *L* to relinquish resource *R*.



priority inversion

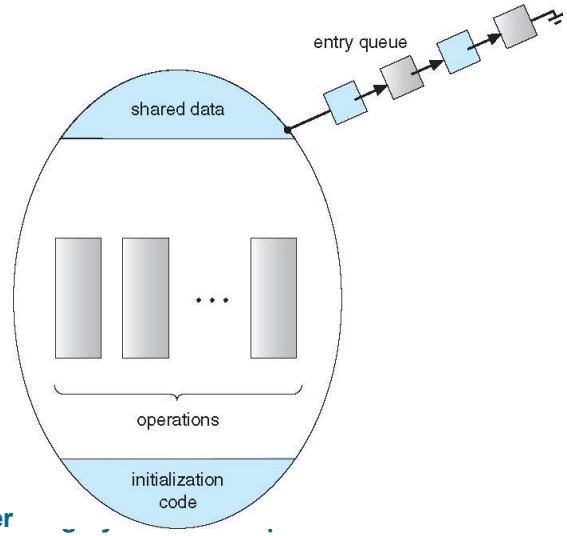
unbounded priority inversion

Deadlock and Starvation

- Solution
  - Use priority-inheritance protocol
    - All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.
    - When they are finished, their priorities revert to their original values.
  - This protocol solve the previous priority inversion problem

Monitors

- Monitor (管程)
  - A high-level abstraction that provides a convenient and effective mechanism for process synchronization
    - *Abstract data type*, internal variables only accessible via procedures
    - Only one process may be active within the monitor at a time
    - Can utilize condition variables to suspend or resume processes



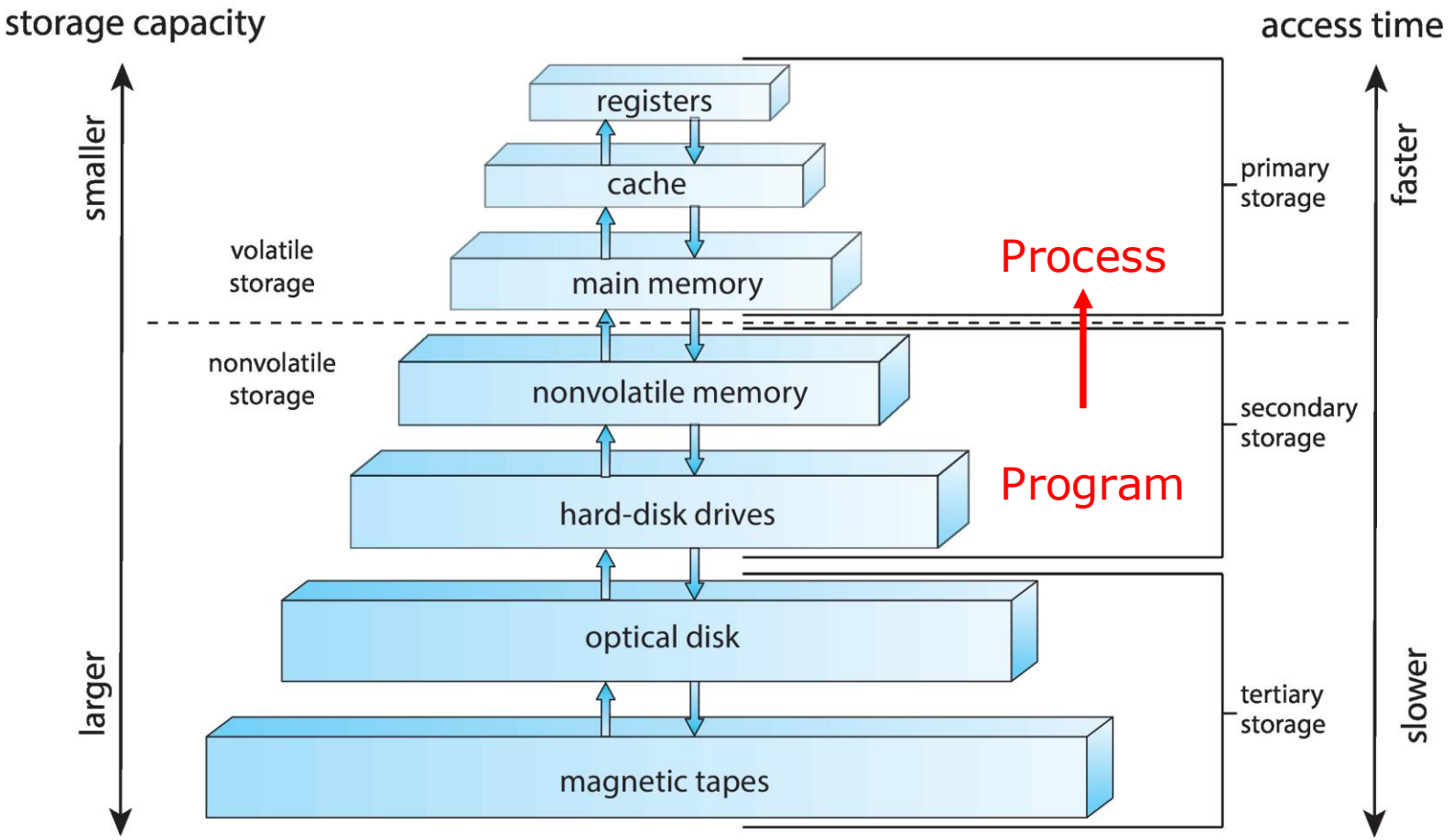

```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (…) { …. }
        procedure Pn (…) {……}
        Initialization code (…) { … }
}
```

Chapter 9: Main Memory

Outline

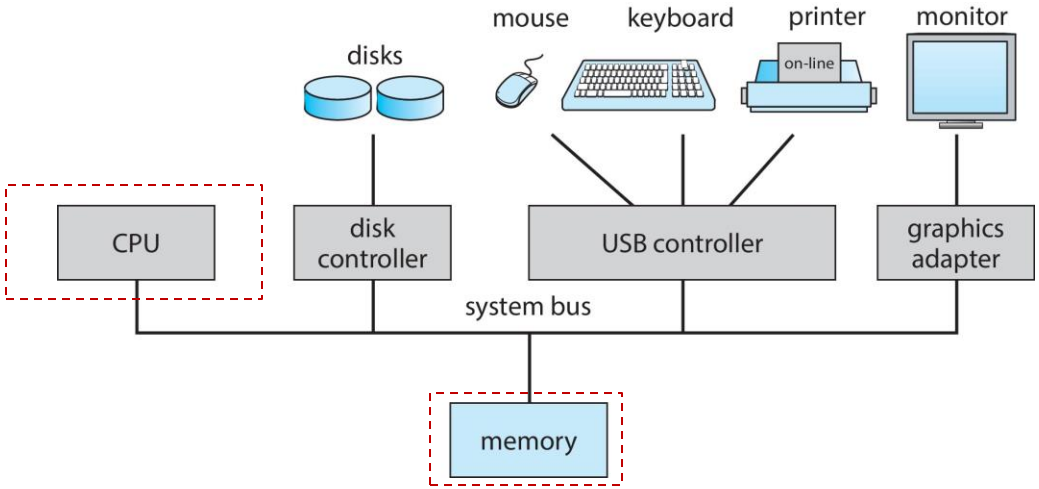
- Background
- Memory Allocation
  - Contiguous Memory Allocation
  - Frame
    - Page Table
    - Address Translation
  - Page Table Schema
    - Hierarchical Page Table
    - Hash Table
    - Inverted Table
  - Examples

Review: Storage Hierarchy



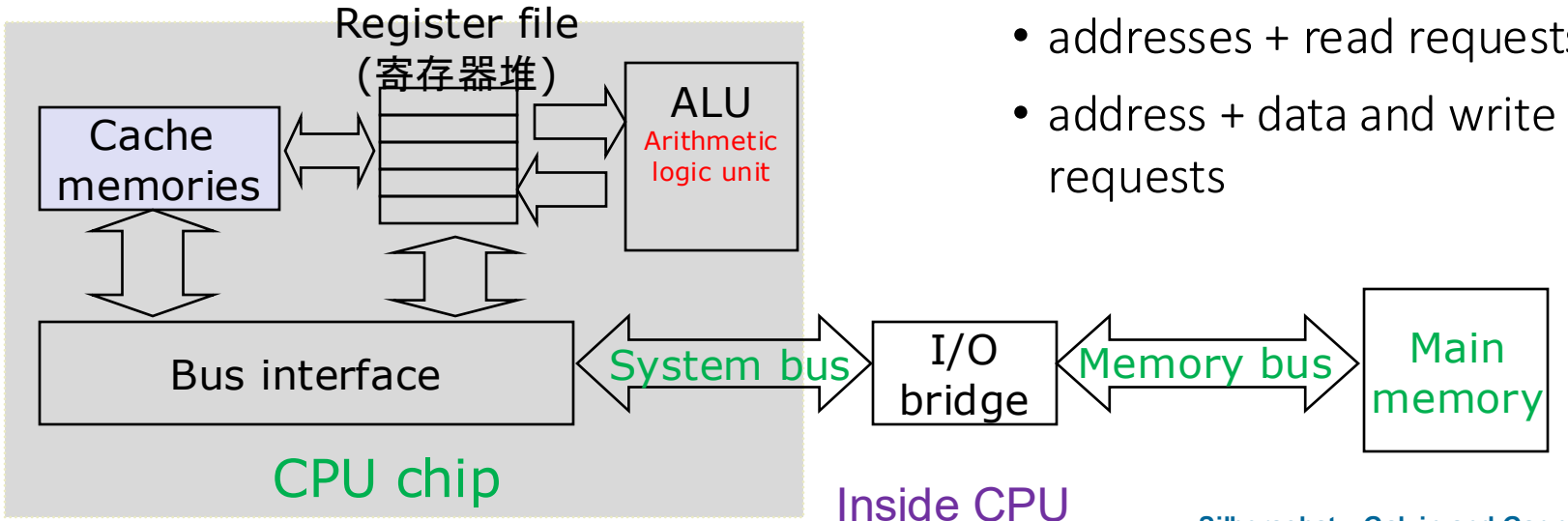
Then, how CPU accesses the process in the memory?

Relationship of CPU and Memory



A PC System Structure

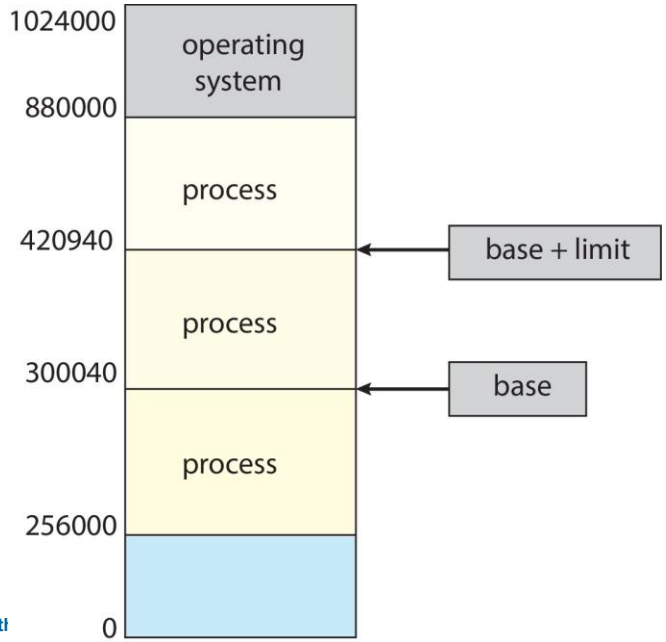
- Main memory and registers are only storage that CPU can access directly
- Register access is done in one CPU clock (or less)
- Main memory can take many cycles, causing a stall (delay)
- Memory unit only sees from CPU a stream of:
  - addresses + read requests, or
  - address + data and write requests



Inside CPU

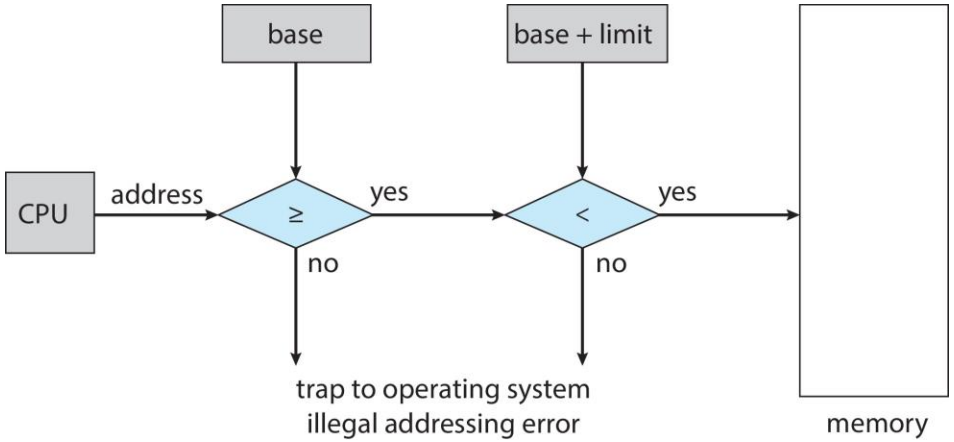
Protection

- CPU notifies memory where to read or write
  - Is this "where" reliable?
  - Will the instructions go to other process's land?
- Protection is needed
  - An old but simple method
    - a pair of base register and limit register for each process to address of the process



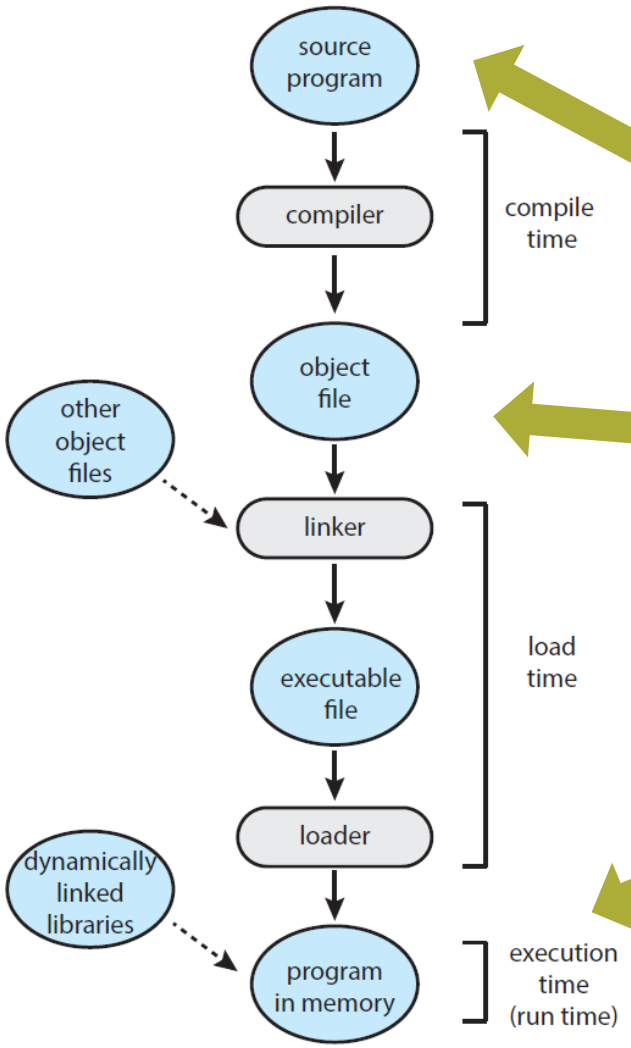
Hardware Address Protection

- Two registers are needed
  - Base register
  - Limit register
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- The instructions to load base and limit registers are privileged
  - No process can spy on another one by changing its own base register to be the same as other's.

Address Binding in A Typical Multistep Processing of a User Program



- How are process addresses determined in memory?
- Addresses are represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
    - i.e. int count; &count
  - Compiled code addresses bind to relocatable addresses (addresses relative to the start of the code).
    - i.e. "14 bytes from beginning of this code module"
  - Linker or loader will bind relocatable addresses to absolute addresses (addresses relative to the lowest address 0000 in memory).
    - i.e. 74014
- Each binding maps one address space to another address space

Linking: Dynamic vs Static

- Static linking –system libraries and program code are combined by the loader into the binary image
  - Every program includes library: wastes memory
- Dynamic linking –linking is postponed until execution/run time
  - Operating system checks if routine (子程序) is in process memory address, If not in address space, add to address space
  - Small piece of code, stub (存根), is used to locate the appropriate memory -resident library routine
    - Stub replaces itself with the address of the routine and executes the routine
  - Particularly useful for libraries which are used by many processes
    - Keep only one copy of the library in memory which is shared by all the processes.
    - known as shared libraries (.dll on Windows and .so on Linux, and .a is Linux static library)

Binding of Instructions and Data to Memory

- Types of systems determine when the binding to memory address happens
  - Compile time: Absolute address is used in code
    - Example: embedded systems with no kernel.
    - Disadvantage: code must be re-compiled if starting location in memory changes
  - Load time: Relocatable code is generated if memory location is not known at compile time
    - Example: very old multiprogramming systems
    - Program's code has a relocation table of relative addresses of all the things in the code
    - The real memory address is calculated according to the given location.
    - Disadvantage: the address in memory is fixed after the program is loaded into memory.

Binding of Instructions and Data to Memory

- Types of systems determine when the binding to memory address happens
  - Execution time: Binding is delayed until run time.
    - Example : most computers today
    - A process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps
      - e.g., MMU

This lecture talks about the binding in execution time.

Logical vs. Physical Address Space

- Memory management is to
  - bind a logical address space to a separate physical address space
    - Logical address
      - generated by the Program/CPU
      - also referred to as virtual address
      - all the addresses used by the CPU
      - Addressed in bytes
      - Size of space depends on the CPU bits
    - Physical address
      - all the addresses used by the memory hardware
      - Addressed in bytes
      - Size of space depends on real size of physical memory

Thinking:
If a computer has 32-bit CPU and memory address is 16-bit, how big is the logical address space and the physical memory address?

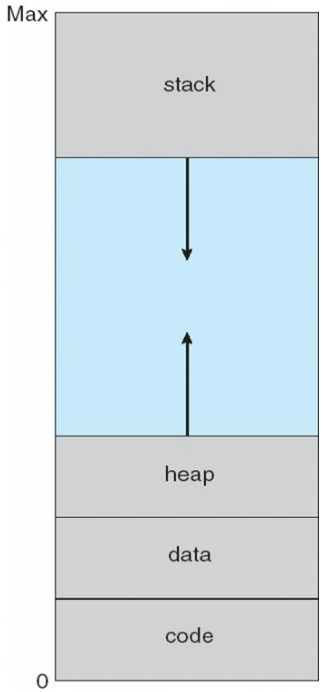
Logical vs. Physical Address Space

- Logical and physical addresses are the same in
  - compile-time address binding, and
  - load-time address-binding schemes
- logical (virtual) and physical addresses differ in
  - **execution-time address-binding scheme facilitates swapping**

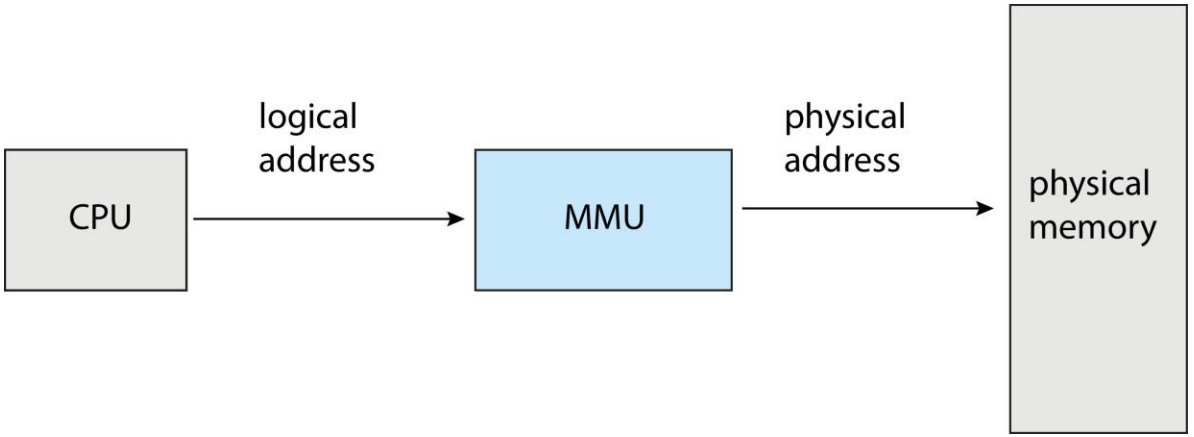
This lecture talks about the binding of logical and physical addresses at execution time.

Memory-Management Unit (MMU)

- MMU
  - Memory Management Unit
  - Hardware device that at run time maps (translates) logical (virtual) address to physical address

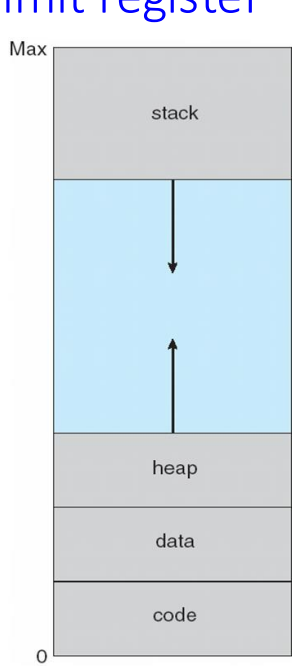


Logical address space

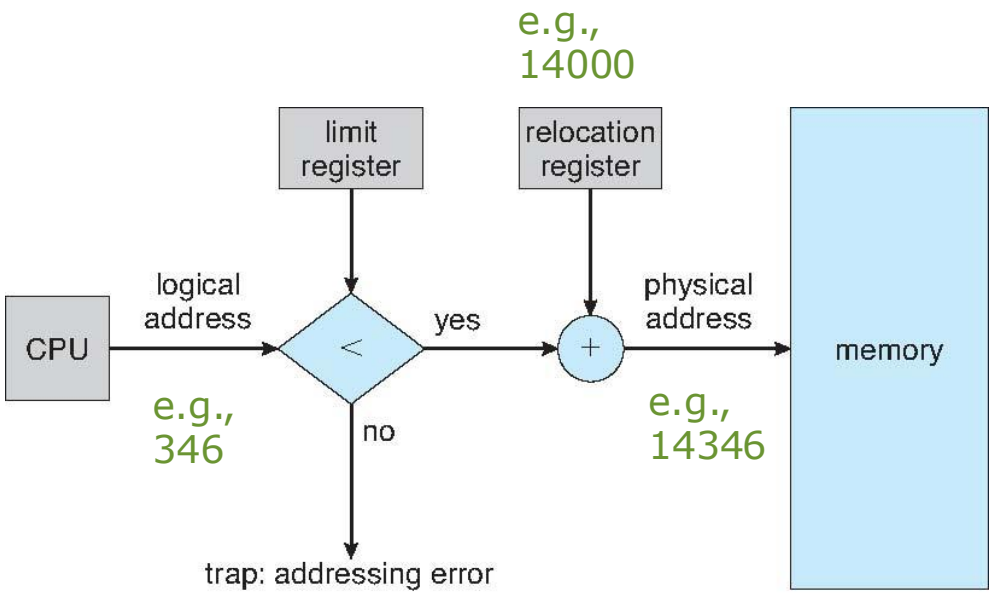


Memory-Management Unit (Cont.)

- A simple mapping scheme
  - relocation register
  - limit register



Logical address space



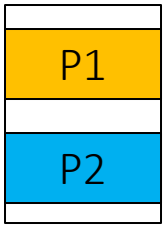
- Logical addresses of all processes start at 0000
- Two processes can use the same logical addresses, these addresses will be mapped to different places in the physical memory
- The kernel changes the relocation and limit register during a context switch

Dynamic Loading

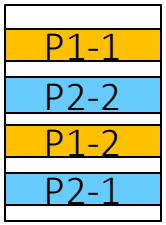
- Should the entire program be loaded in to the memory to start the execution?
  - No!
  - Code is loaded from the hard disk into memory only when it is needed, on demand
  - Dynamic loading
    - The loading of a process routine when it is called rather than when the process is started.
- Why
  - Better memory-space utilization
  - Unused routine is never loaded
  - Some large amounts of code handle infrequently occurring cases
- How
  - All routines are kept on disk in relocatable load format
  - Implemented through program design by programmers, not OS
  - OS can help by providing libraries to implement dynamic loading

Memory Allocation

- Memory can be shared by many processes, including kernel and user processes
  - It should be allocated efficiently
  - Resident kernel usually is held in low memory
  - User processes is held in high memory
- Memory for each process can be
  - Contiguous (early method)



  - Not contiguous (modern method)

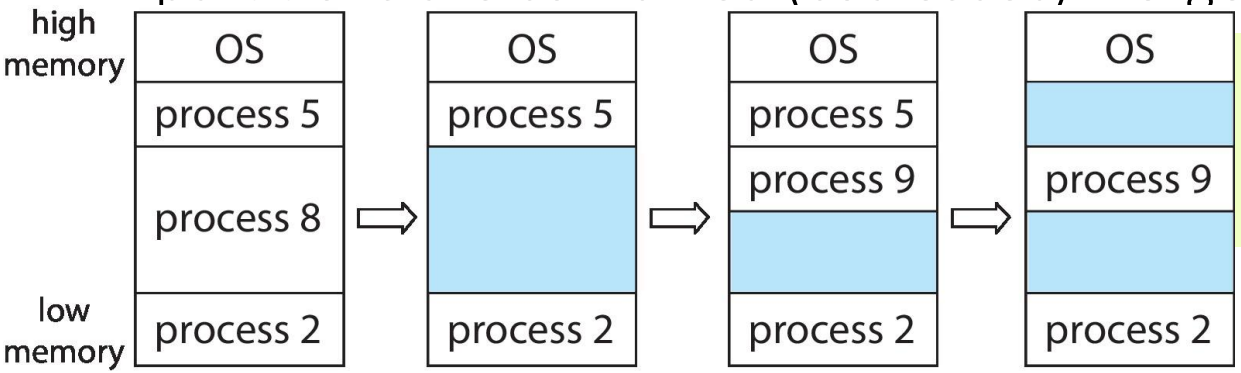


Continuous Partition

- Memory is divided into multiple partitions
  - Size of each partition is fixed
    - Multiple-partition allocation
    - Disadvantages:
      - Degree of multiprogramming is limited by number of partitions
      - Memory is wasted if a process does not fully use its partition
  - Size of each partition is variable
    - Variable-partition
    - Propositional to a given process' size

Continuous Partition: Variable Partition

- Kernel maintains information about
  - allocated partitions
  - free partitions (hole)
- Hole
  - block of available memory
  - holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - When a process exits, its partition is freed, adjacent free partitions are combined (coalesced/merged)



Thinking:
Memory after process 9 is released?

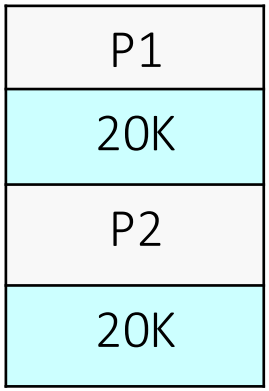
Continuous Partition: Variable Partition

- How to satisfy a request of size *n* from a list of free holes?
- Three allocation algorithms
  - First-fit: Allocate the *first* hole that is big enough
  - Best-fit: Allocate the *smallest* hole that is big enough;
  - Worst-fit: Allocate the *largest* hole
- Comparison of three algorithms
  - Speed
    - First-fit > best-fit and worst-fit
      - Both best-fit and worst-fit need to search the whole list
  - Memory usage
    - Best-fit > first-fit > worst-fit (theoretically)
    - First-fit > best-fit (case studies)
      - Reason: best-fit produces many small holes that can never be used

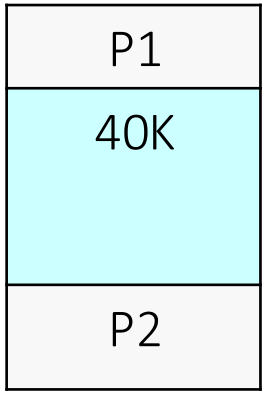
Fragmentation

- External Fragmentation
  - Total memory space exists to satisfy a request, but it is not contiguous
  - Small holes might be wasted because the holes are too small to fit any process.
- Internal Fragmentation
  - Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
  - Any memory which is allocated to a process but that the process did not ask for is wasted.

P3: 31K
20+20>31,
but not contiguous



External fragmentation



P3: 31K
2^5K=32K will be allocated,
but 1k is not used

Internal fragmentation

Fragmentation

- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 50-percent rule
    - the memory lost to fragmentation is about 50% the size of the allocated memory
  - 1/3 of RAM may be unusable

Fragmentation

- Solutions to fragmentation problems
  - External fragmentation
    - Compaction
      - Shuffle memory contents to place all free memory together in one large block
      - possible only if relocation is dynamic, and is done at execution time
      - Pay attention to I/O problem while there is I/O operations on the affected memory
    - Use non-contiguous memory allocation
  - Internal fragmentation
    - No way
    - Memory is allocated in the block of 2^n bytes, rather than byte by byte.

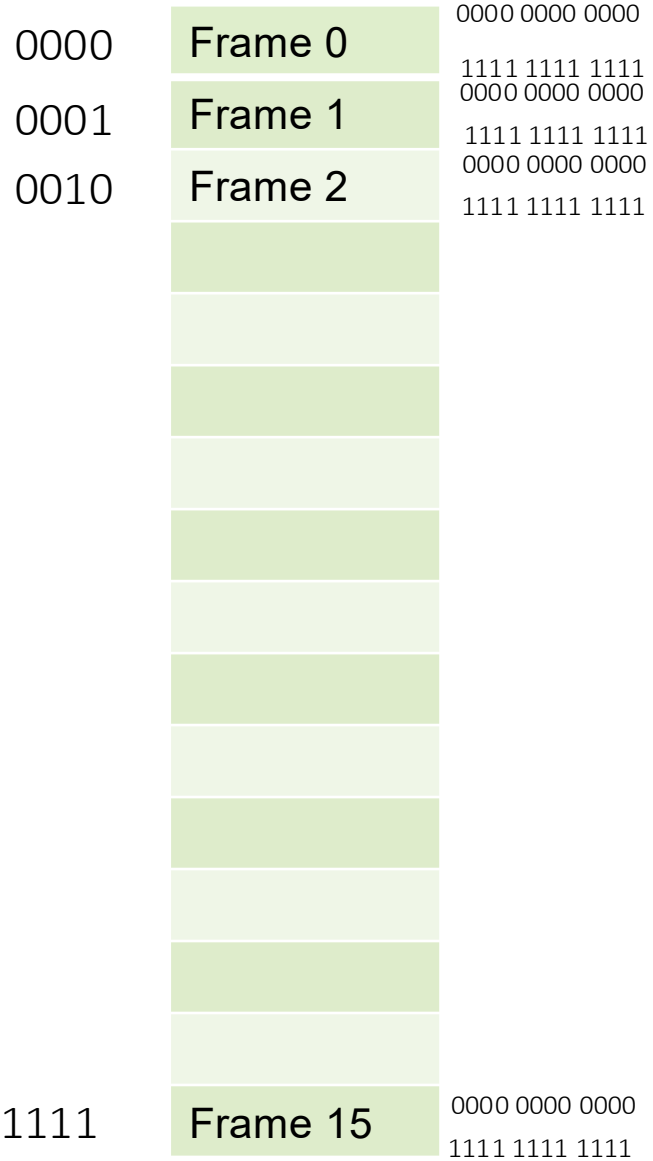
Non-continuous Partition

- Physical address space of a process can be noncontiguous
- Process is allocated physical memory whenever the total free memory blocks are enough
  - Avoids external fragmentation
- Frame
  - Divide physical memory into fixed-sized blocks
  - Each block is called a frame
  - The size of each frame is power of 2, between 512 bytes and 16 Mbytes
  - Kernel keeps track of all free frames

Thinking:

If RAM's address uses 16 bits, each frame is 4K, how many frames are in the memory?

Non-continuous Partition



16 bits: 2^{16} , 4K: $4*2^{10} = 2^{12}$

Total frames: $2^{16}/ 2^{12}= 2^{4}$

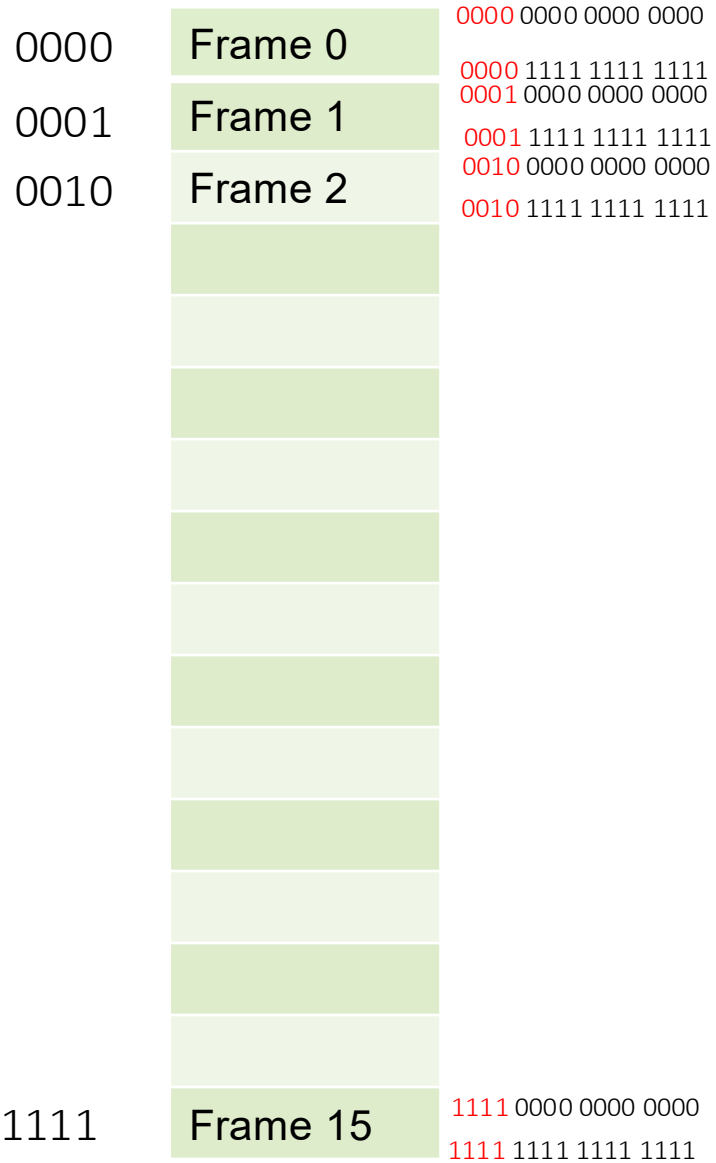
Frame number

Offset in each frame

Thinking:

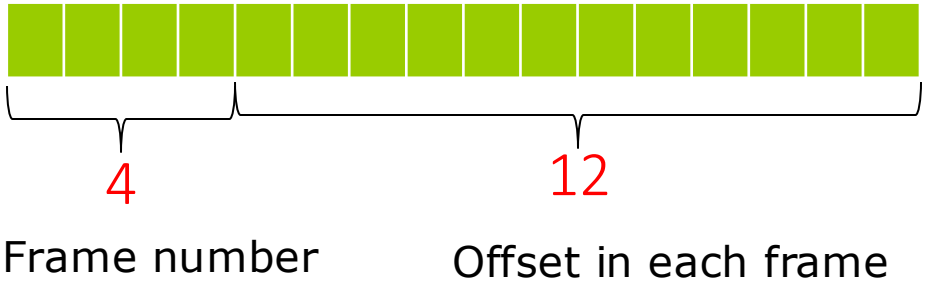
If RAM's address uses 16 bits, each frame is 4K, how many frames are in the memory?

Non-continuous Partition



16 bits: 2^{16} , 4K: $4*2^{10} = 2^{12}$

Total frames: $2^{16}/ 2^{12} = 2^4$



Thinking 1:

If a datum is in the address

0100 0000 0000 0001?

Which frame is it in? where is it in the frame?

Thinking 2:

If a process's size is 8K + 1, how many frames are allocated? Is there internal fragmentation?

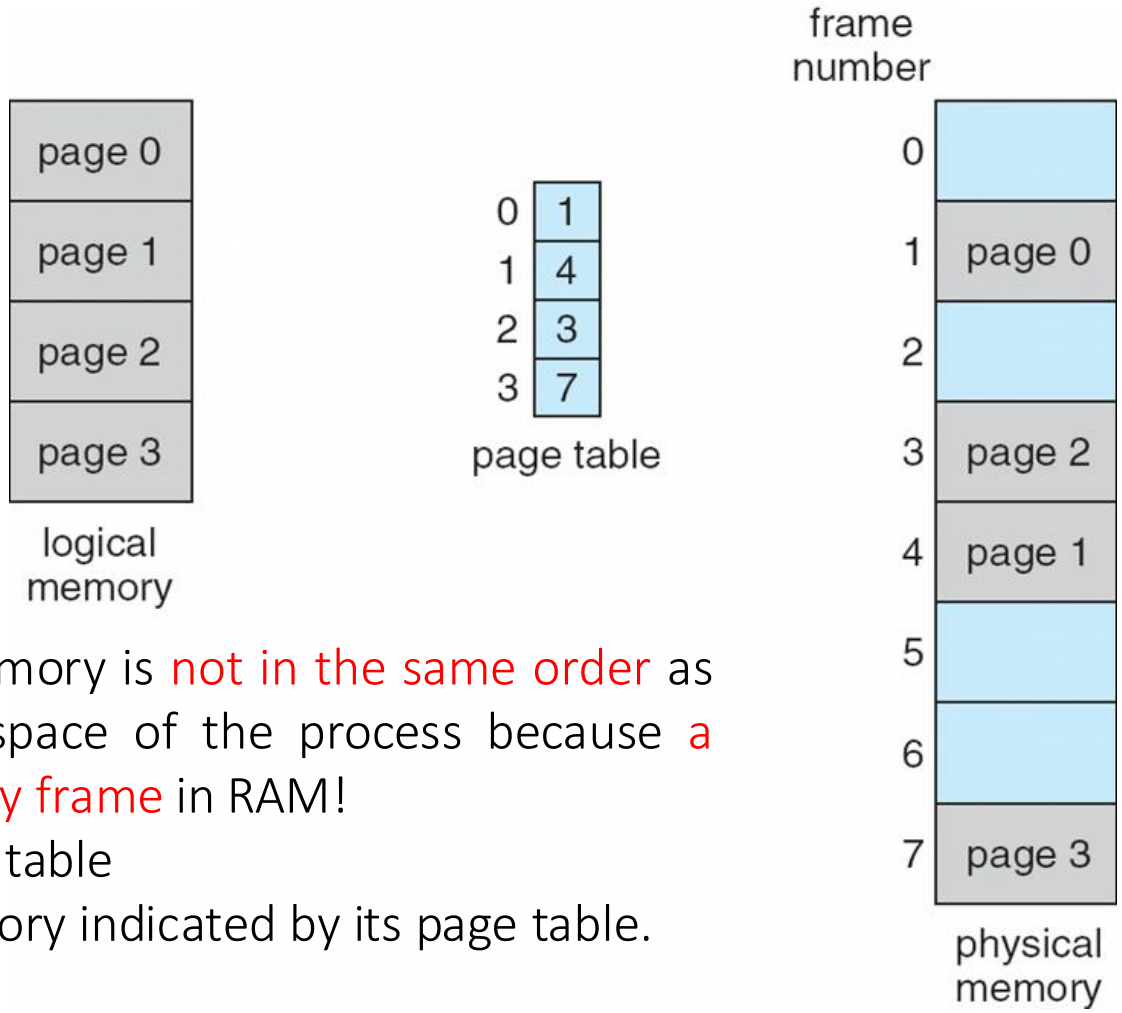
Non-continuous Partition

- How a logical address maps to a physical address?
- Pages
  - Divide logical memory into blocks of same size called pages
  - Page size = frame size
  - To run a program of size *N* pages, need to find *N* free frames and load program
  - Frames can be anywhere(non-contiguous) in RAM!
- A page table is needed to translate logical to physical addresses

Thinking:

If logical space uses 32 bits, each frame is 4K, how many pages are there in logical space? $2^{32}/ 2^{12}= 2^{20}$

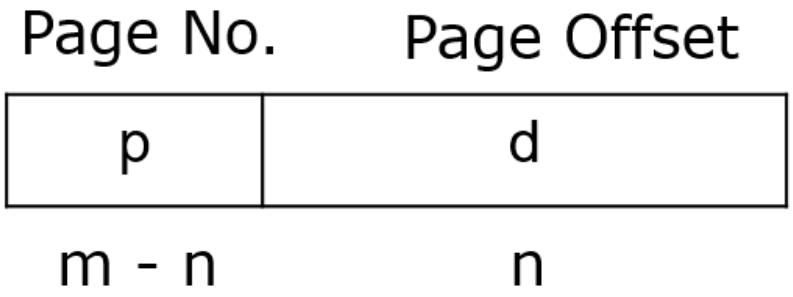
Address Translation Scheme: Page Table



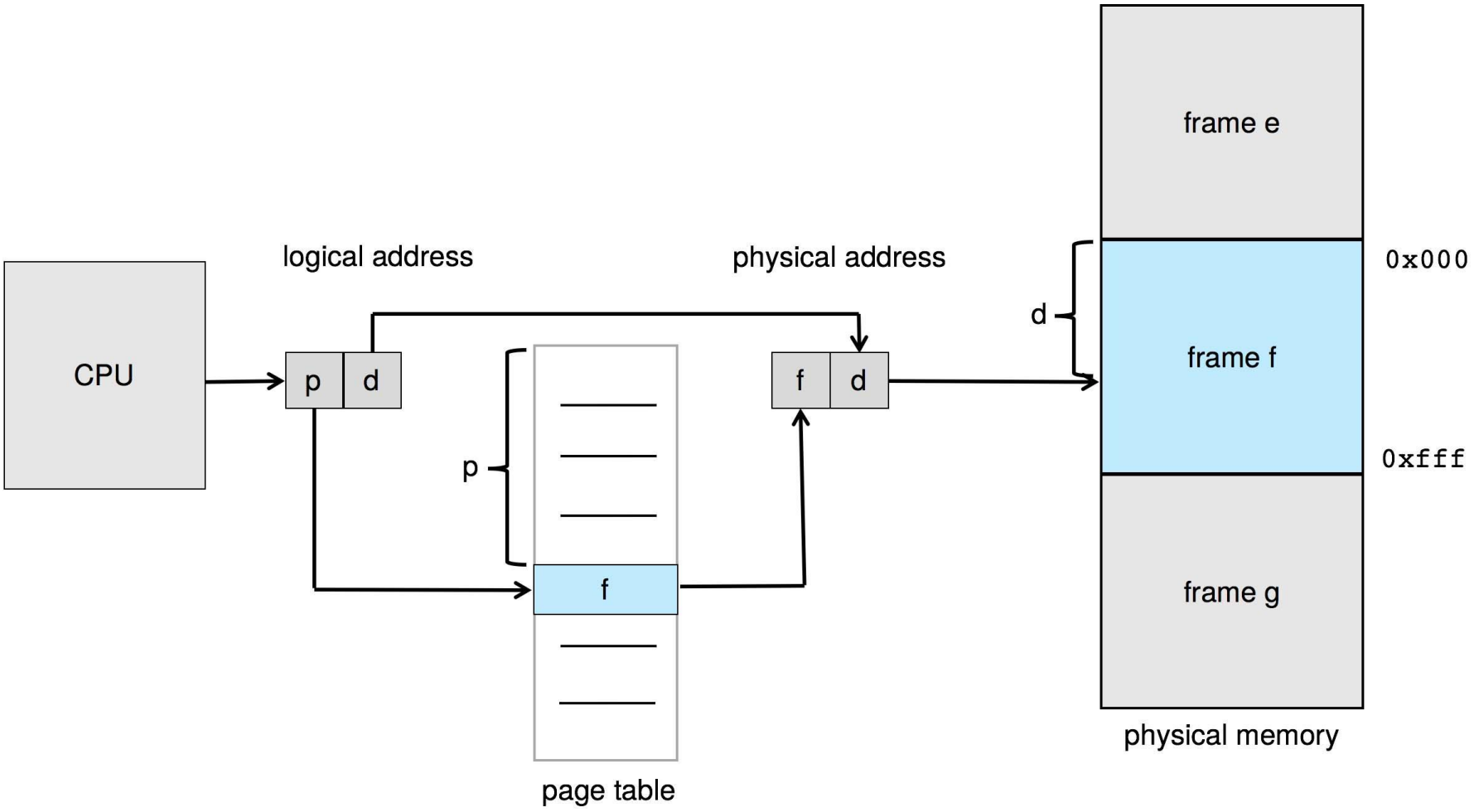
- The data in physical memory is not in the same order as in the logical address space of the process because a page can be stored in any frame in RAM!
- Each process has a page table
- Process access the memory indicated by its page table.
- Page table is
  - part of the process's PCB.
  - invisible to the process itself.

Address Translation Scheme: Page Table

- Given a logical address space 2^m and page (frame) size 2^n, there are $2^m/2^n = 2^{m-n}$ pages in a process's logical address space
- Logical address generated by CPU (m bits) is divided into
  - Page number (p) – used as an index into a page table which contains base address (frame number) of each page in physical memory
  - Page offset (d) – combined with base address (frame number) to define the physical memory address that is sent to the memory unit



Paging Hardware



Page number *p* to frame number *f* translation

TLB

- TLB table

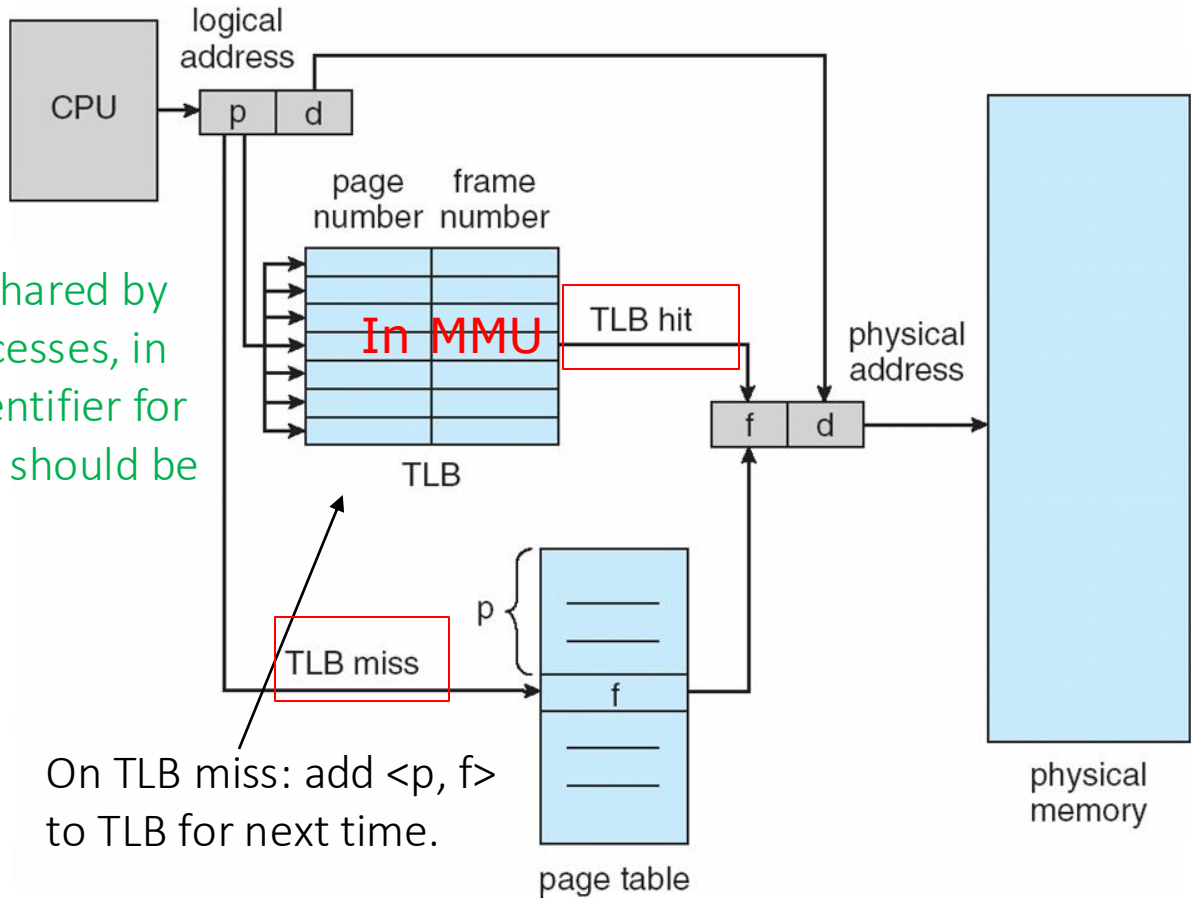
Page #	Frame #

- Given a logical page number *p*
  - If *p* is in TLB (called TLB hit)
    - get corresponding frame number from it
  - If *p* is not in the TLB table (called TLB miss)
    - get frame number from page table in memory
    - then update the TLB to contain this page # and frame #
      - Reason
        - » if that page was just used by the process, it is probably going to be used again soon.

TLB

- TLB is
  - typically small
    - 64 to 1,024 entries
  - Shared by multiple processes
    - Store address-space identifiers (ASIDs) in each TLB entry to uniquely identify each process to provide address-space protection for each process
    - Reduce the fresh time at every context switch between processes
- How is a TLB entry replaced by a new page-frame pair?
  - Replacement policies must be considered
  - Some entries can be wired down (never be replaced) for permanent fast access (for key kernel code)

Paging Hardware With TLB



TLB: can be shared by multiple processes, in that case, identifier for each process should be added.

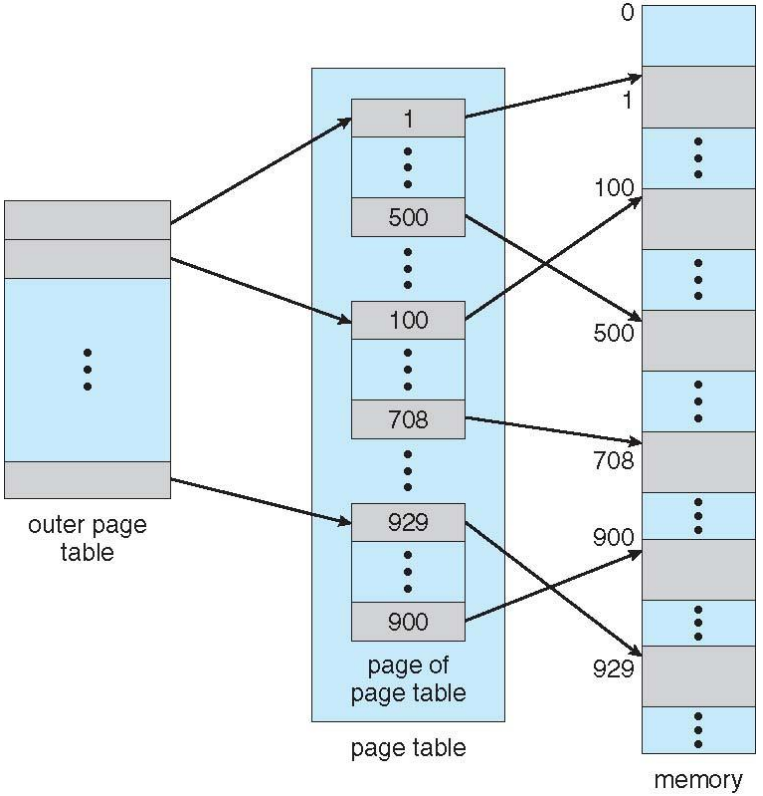
Page table: own by each process

Effective Access Time: An Example

- Hit ratio α is
  - percentage of times that a page number is found in the TLB
- Effective memory access time
  - EAT = α x (c+m)+(1- α) x (c+2m)
  - c: TLB access time, m: memory access time
- An example
  - α: 80%, c: 10 nanoseconds, m: 90 nanoseconds
    - EAT = 0.80 x (10+90) + 0.20 x (10+90 x2) = 118 ns
    - 18% slowdown in access time
- If α: 99%
  - EAT = 0.99 x (10+90) + 0.01 x (10+90 x2) = 100.9ns
  - only 0.9% slowdown in access time
- Big table solutions
  - Hierarchical paging, Hashed page tables, Inverted page tables

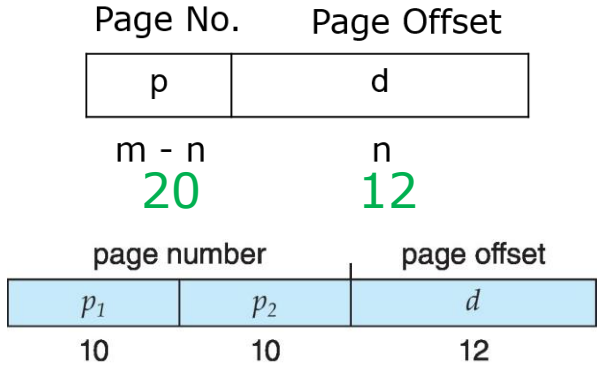
Hierarchical Page Tables

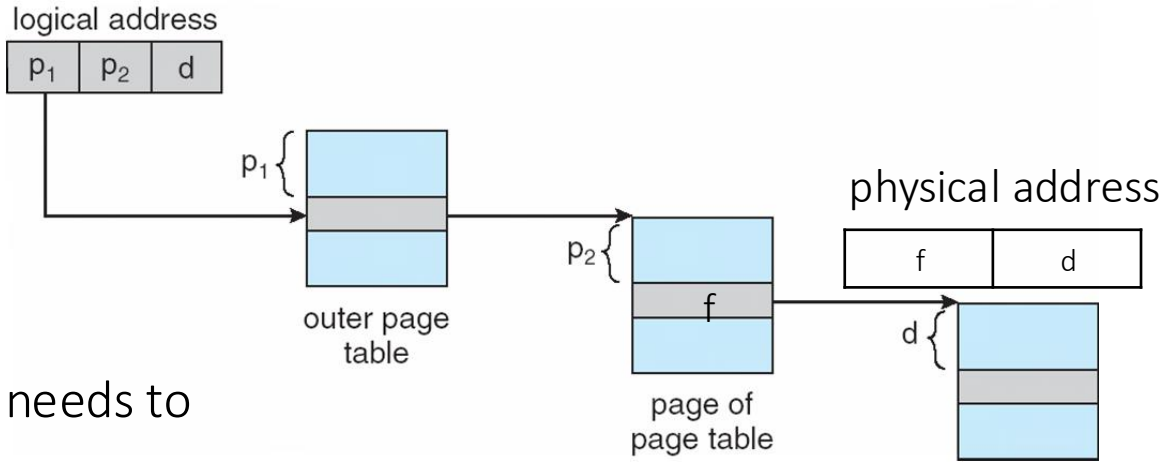
- Split one (big) page table into multiple (small) page tables
  - two-level page table
    - Outer page table
      - Each entry indicates the position of small page table
    - Inner (small) page tables
      - Each small page table (also called page of page table) has the page size
      - Different small page tables can be put in different places
      - Only the part of the page table that the process requires right now needs to be in memory!



Two-Level Paging Example

- logical address space: 32-bit (m=32), page size: 4 KB (n=12)
  - Originally
    - page table length: $2^{m-n} = 2^{20}$
  - Two levels
  - p_1 : a 10-bit page number (1KB)
    - an index into the outer page table
  - p_2 : a 10-bit page offset (1KB)
    - the displacement within the page of the inner page table

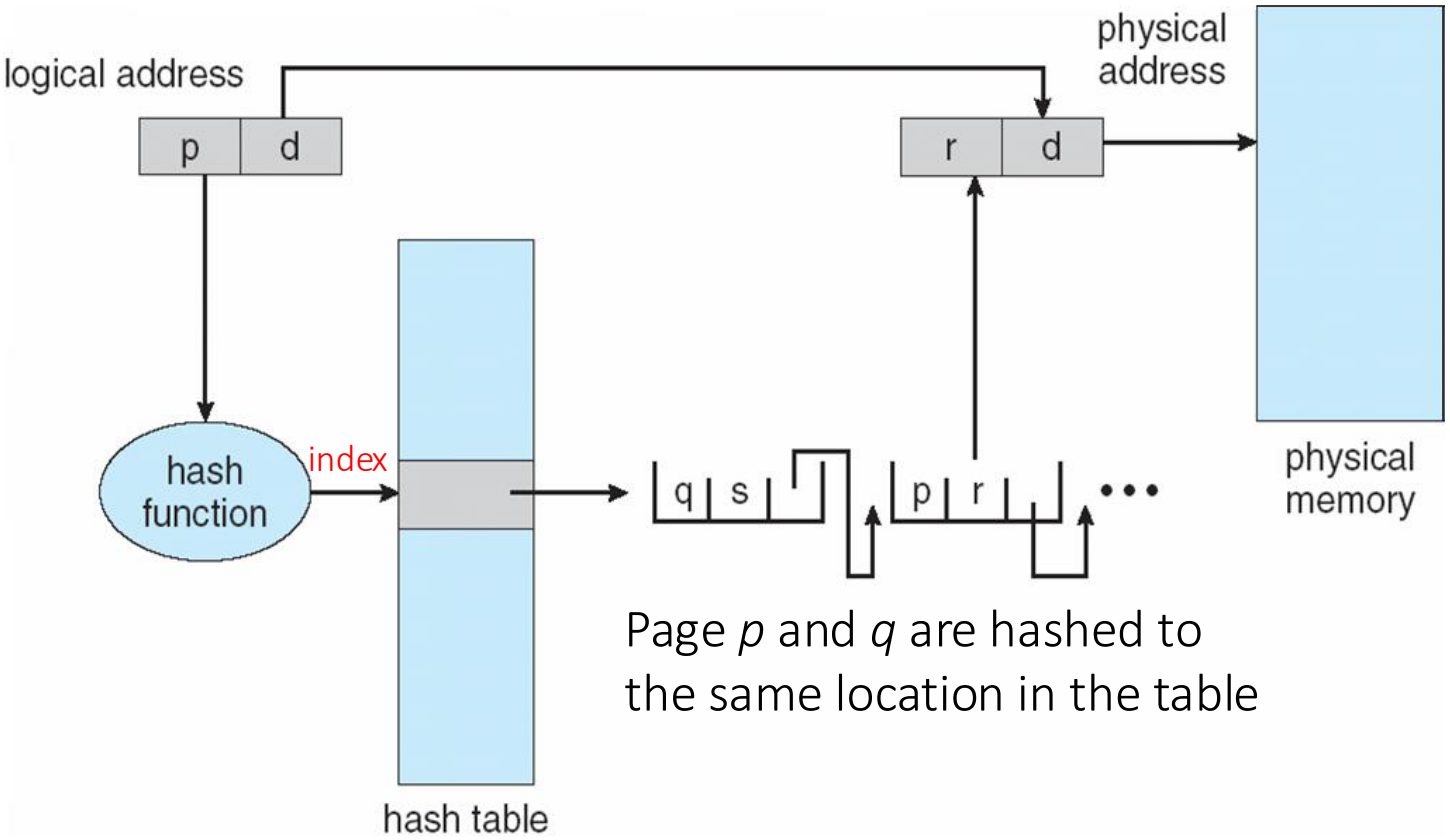




- Advantage: save memory
  - only part of the page table needs to be in memory
- Disadvantage : three memory access

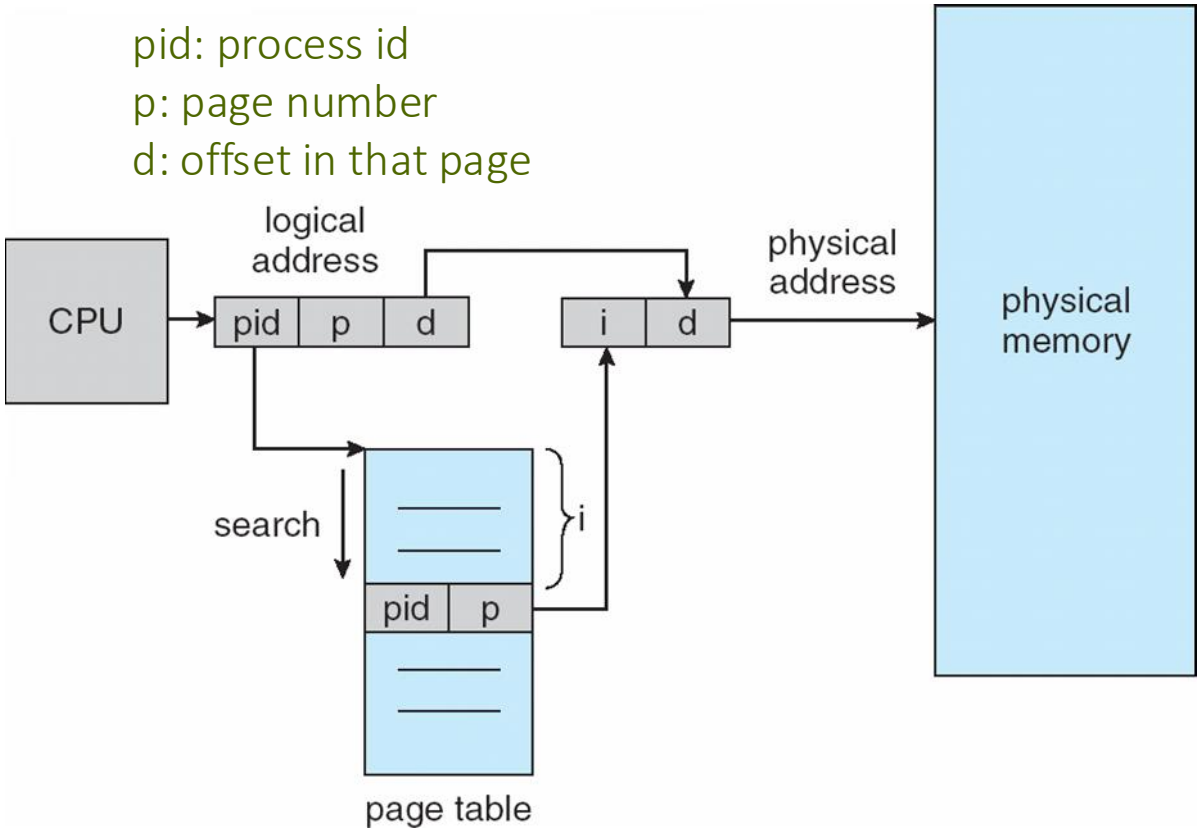
Address translation process

Hashed Page Tables



- An entry in the hash table contains a chain of elements hashing to the same location
- Each element contains
  1. the virtual page number (e.g., *q*)
  2. the value of the mapped page frame (e.g., *s*)
  3. a pointer to the next element.
- This method is commonly used for the address with more than 32 bits

Inverted Page Table



- The length of page table is the same as the size of physical memory
- All processes share one page table
- The page table is actually a reflection of layout of the physical memory frames

Chapter 10: Virtual Memory

Outline

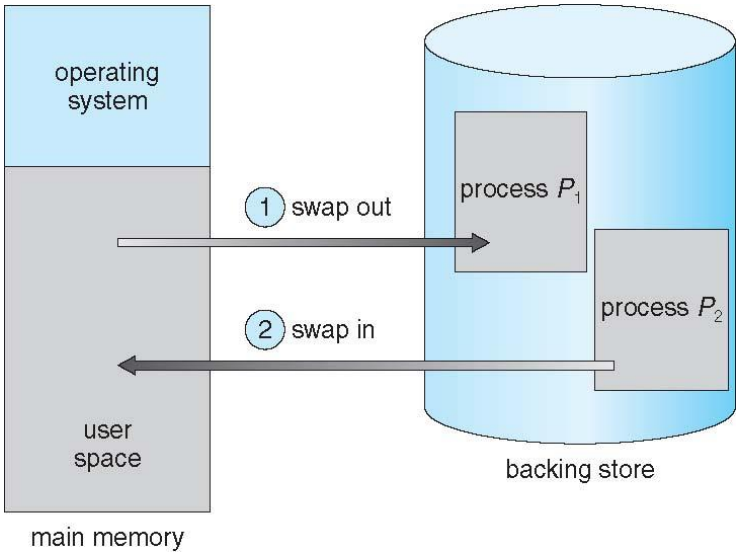
- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Allocating Kernel Memory
- Other Considerations
- Example

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model
- To explore how kernel memory is managed

Multiprogramming

- Multiprogramming
  - Multiple processes can reside in the memory at the same time
- Problem: if a program needs to be run, but there is no enough free memory
- One solution：swap out (换出) a process and swap in (换入) the target process



Backing store（后备存储器）

- is fast hard disk large enough to accommodate copies of all process memory images for all users
- has direct access to memory images

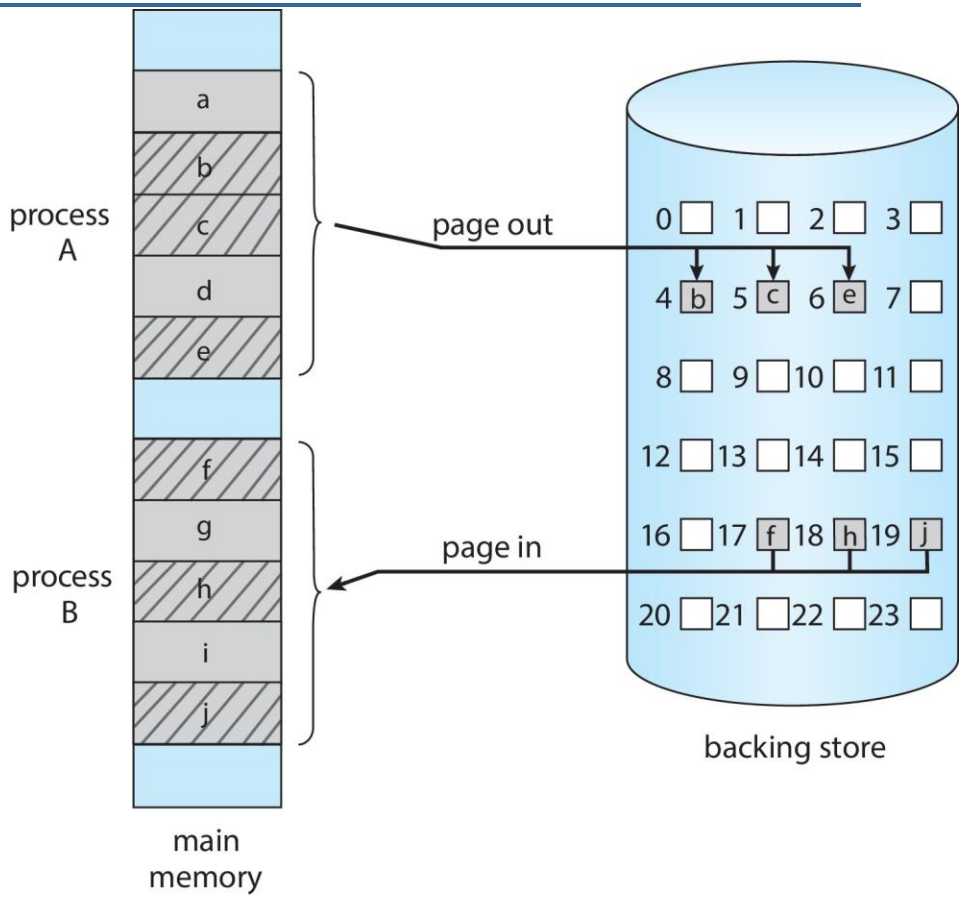
Swapping

- A complete process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- A swapper manipulates entire process swapping
- Advantage
  - Increase multiprogramming
- Disadvantage
  - Context switch time can then be very high
  - Total context switch time includes swapping time for the whole process

Can the swapping time be reduced by reducing the size of memory swapped?

Swapping with Paging

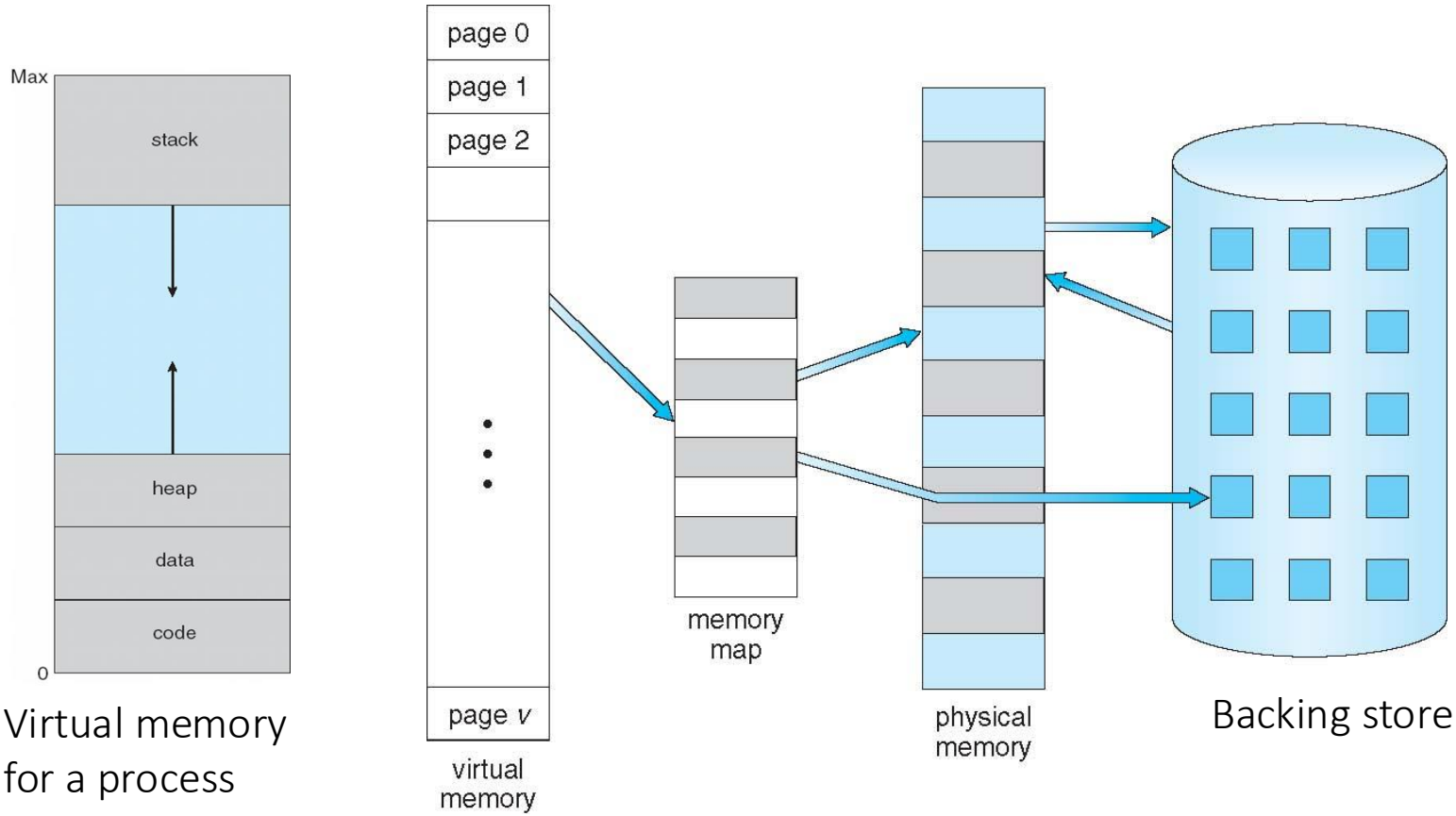
- When memory is low, unused pages are swapped to disk (instead of whole processes)
  - How to determine which pages are unused?
  - What happens when a process suddenly tries to access a page that was swapped out?



Background

- Program can be partially loaded into memory
  - Error handling code, unusual routines, large data structures may not be needed for most of time
- Program length is no longer constrained by limits of physical memory
  - Each program takes less memory → more programs run at the same time
  - Less I/O needed to load programs into memory → each user program runs faster
- Virtual memory – separation of user logical memory from physical memory
  - Logical address space can therefore be much larger than physical address space
  - Only needed pages are loaded into memory

Virtual Memory That is Larger Than Physical Memory

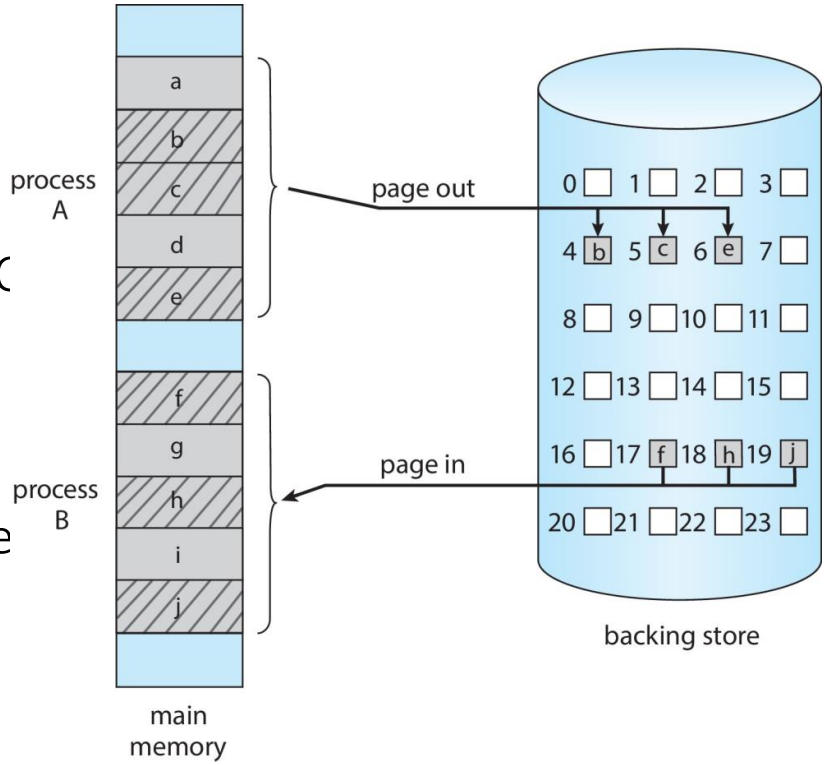


On a modern computer,

- the logical address space (virtual memory) is 2^{64} bytes = 16 billion GB, but
- the physical address space (RAM) is only a few GB (a few hundred GB for the biggest computers on earth).

Demand Paging

- Demand paging
  - Could bring a page into memory only when it is needed
  - Advantages:
    - Less I/O needed, no unnecessary I/O
    - Less memory needed
    - Faster response
    - More users/processes
- Page is needed (called a reference to the page
  - invalid reference might be caused by
    - wrong address ⇒ process abort
    - not-in-memory ⇒ bring to memory
- A pager can swap in and out one page at a time (not a whole process like swapper does).

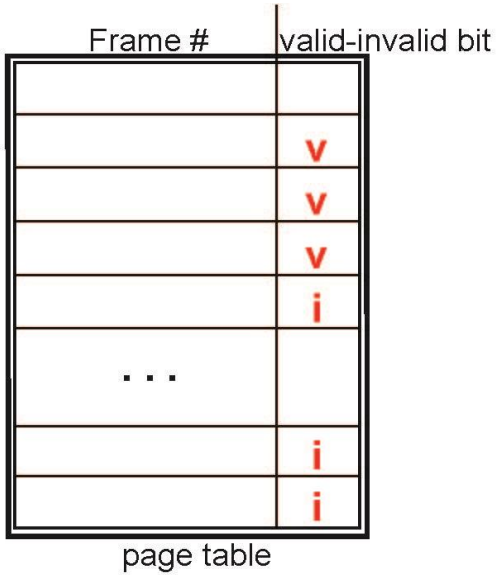


Demand Paging: Basic Concepts

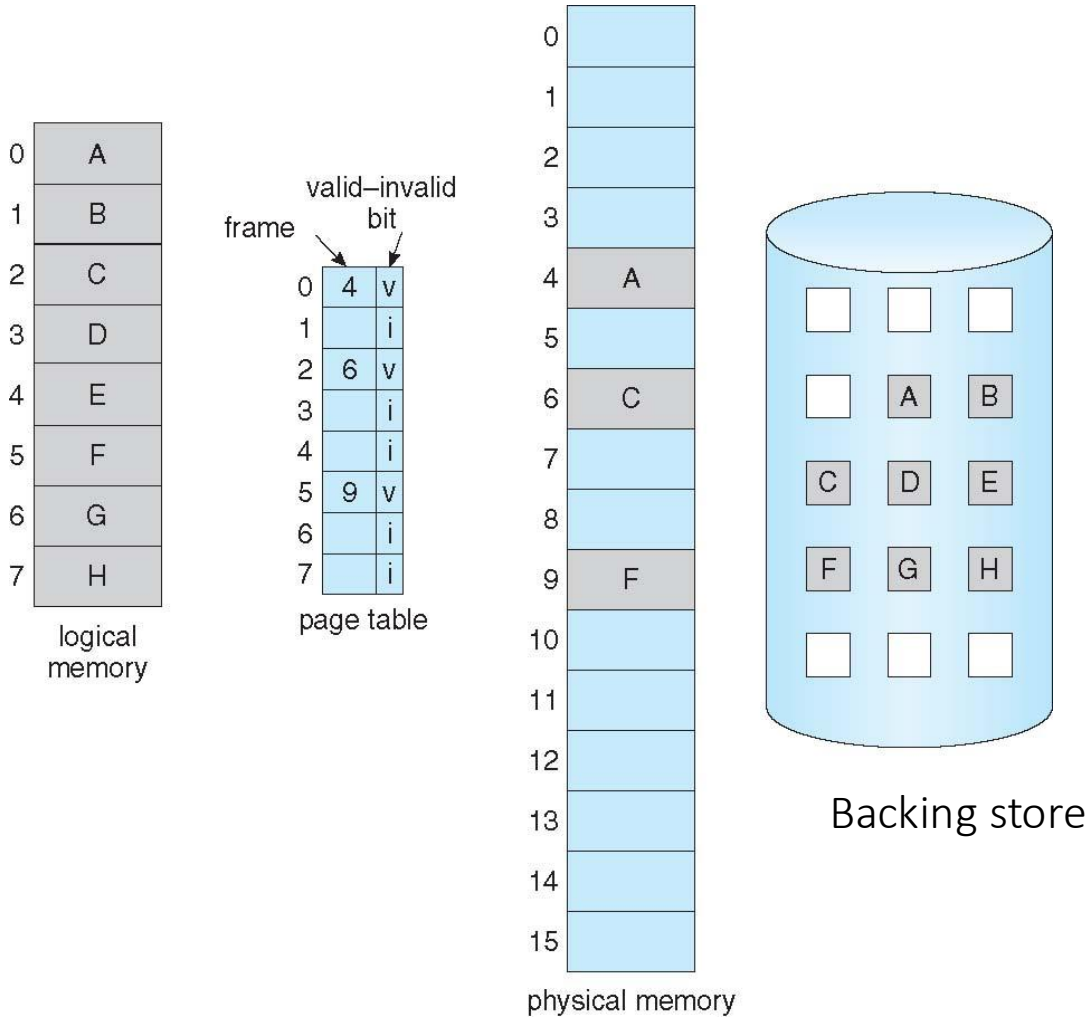
- The pager brings in only those pages into memory that the process actually wants to use
  - Question: How to determine that set of pages?
    - Hardware support: MMU
      - If pages needed are already memory resident
        - » No difference from non-demand paging
      - If pages needed are not in memory
        - » Need to detect and load the pages into memory from storage

Valid-Invalid Bit

- To check if a reference is valid or not, associate each page table entry with a valid–invalid bit
  - *v*: the page is in-memory
  - *i* : the page is
    - either not-in-memory, or
    - a wrong logical address
- Initially valid–invalid bit is set to *i* on all entries



An Example: Some Pages Are Not in Main Memory



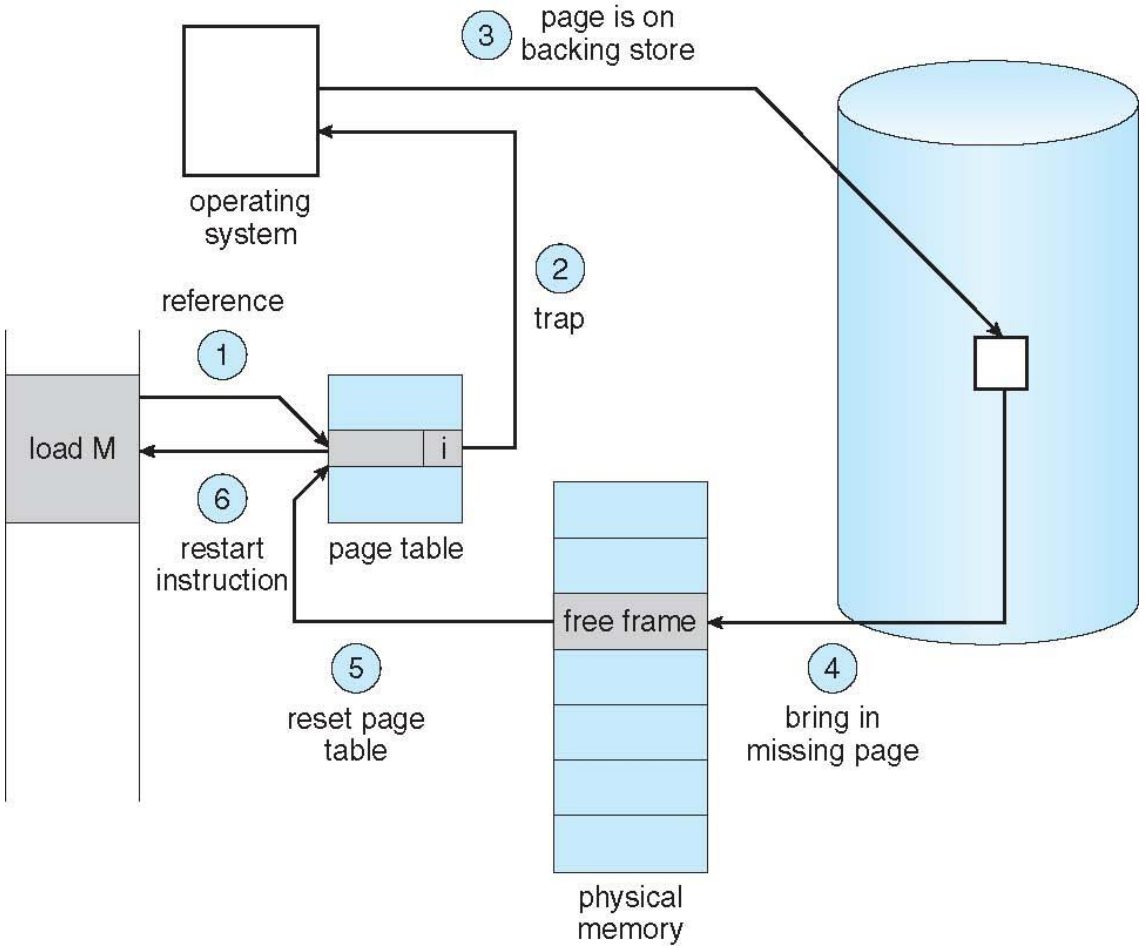
Pages 0, 2, and 5 are in the memory

Pages 1, 3, 4, 6, and 7 are not in the memory

Page Fault

- Page fault
  - Caused when a process makes a reference to a page which is not in RAM,
  - MMU detects that the page is marked as invalid in the process's page table,
- Two possibilities
  - the process is trying to access a page that does not exist at all
    - E.g., the process has pages for 1 to 10, but CPU wants to access page 12
    - Solution: Process aborts
  - The page is not in memory but is somewhere on the hard disk
    - E.g., the process has pages from 1 to 10, CPU wants to access page 2, but this page has not been loaded into RAM
    - Solution: Load the page from hard disk to RAM

Steps in Handling a Page Fault

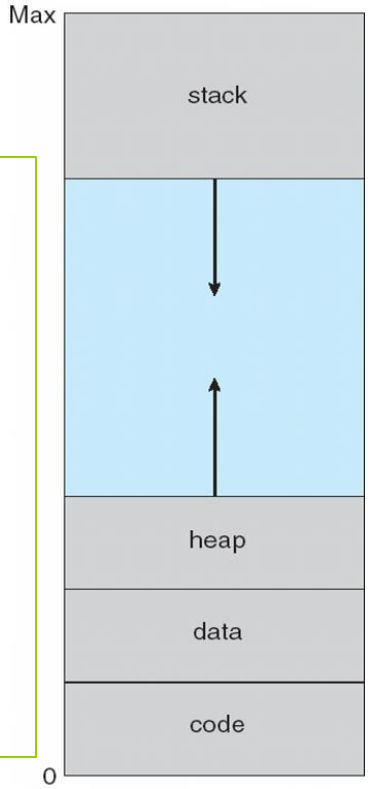


1. CPU makes a reference to a page which is marked *i* in the page table
2. MMU traps to kernel
   a) if the reference page does not exist at all, stop the process;
   b) otherwise, go to 3
3. Find the page in the hard backing store
4. Find a free frame in the memory and load the page into that frame
5. Reset the page table such that the valid bit is *v*
6. Restart the process by re-executing the instruction which caused the page fault.

Aspects of Demand Paging

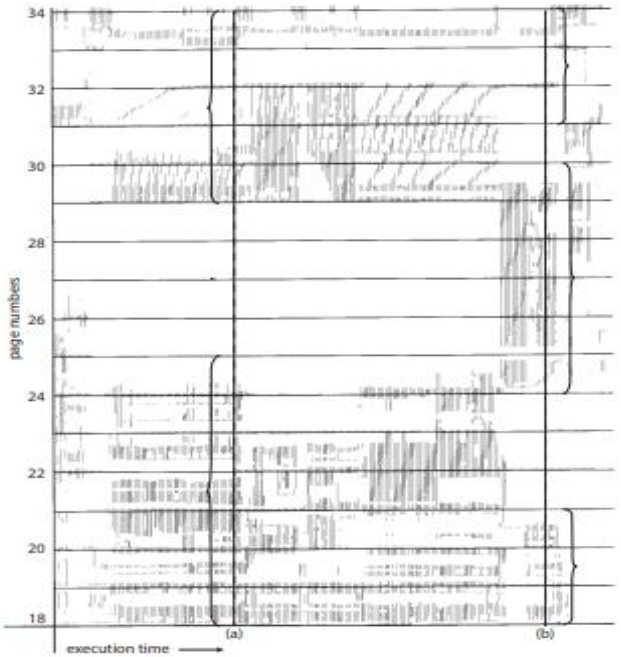
- Pure demand paging
  - Start process with *no* pages in memory (extreme case)
  - Immediate page fault for the first instruction in the process
- A given instruction could access multiple pages -> multiple page faults
  - E.g., suppose the following code

```
int d; //global variable
int main(){
   int i;   //local variable
   int *p;  // pointer
   p = (int*) malloc(sizeof(int))
   … …
   //Thinking: how many pages are usually
   //accessed for the following instruction
   *p= i + d;
}
```



Locality Reference

- Locality of reference
  - The tendency of processes to reference memory in patterns rather than randomly
  - Results in reasonable performance from demand paging

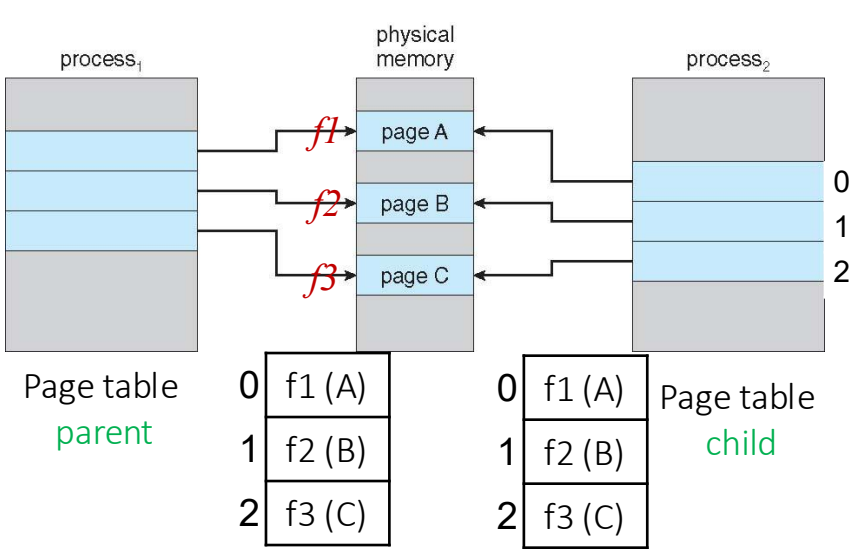


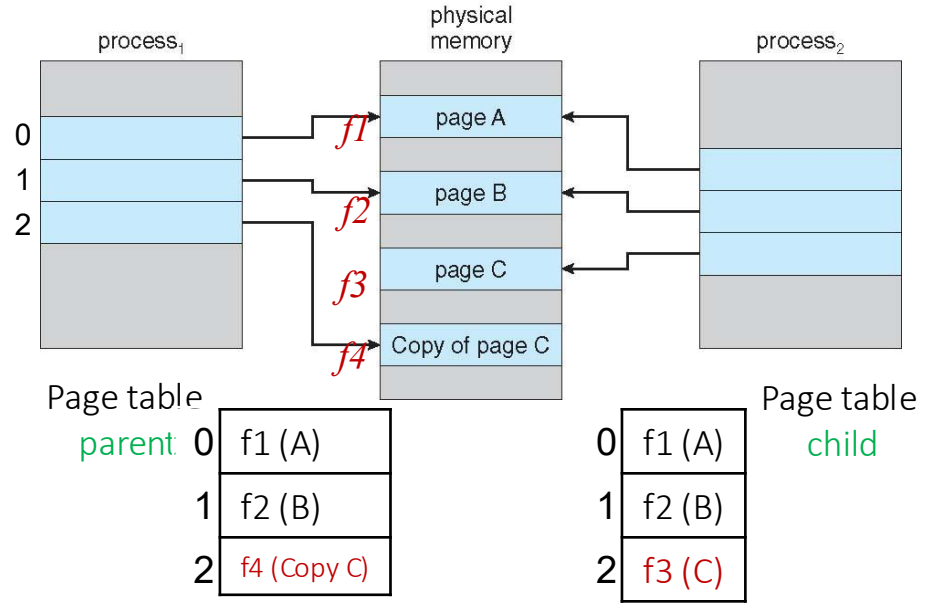
Locality in a memory-reference pattern

- It will fault for the pages in its locality until all these pages are in memory
- It will not fault again until it changes localities
- Process migrates from one locality to another over time
- Localities may overlap (over time)

Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory (vfork instead of fork)
  - When a child process is initially created,
    - only copy the page table of its parent
    - share parents' pages
    - the shared pages are write-protected (using protection bits of page tables).
    - If either process modifies a shared page, then copy that page

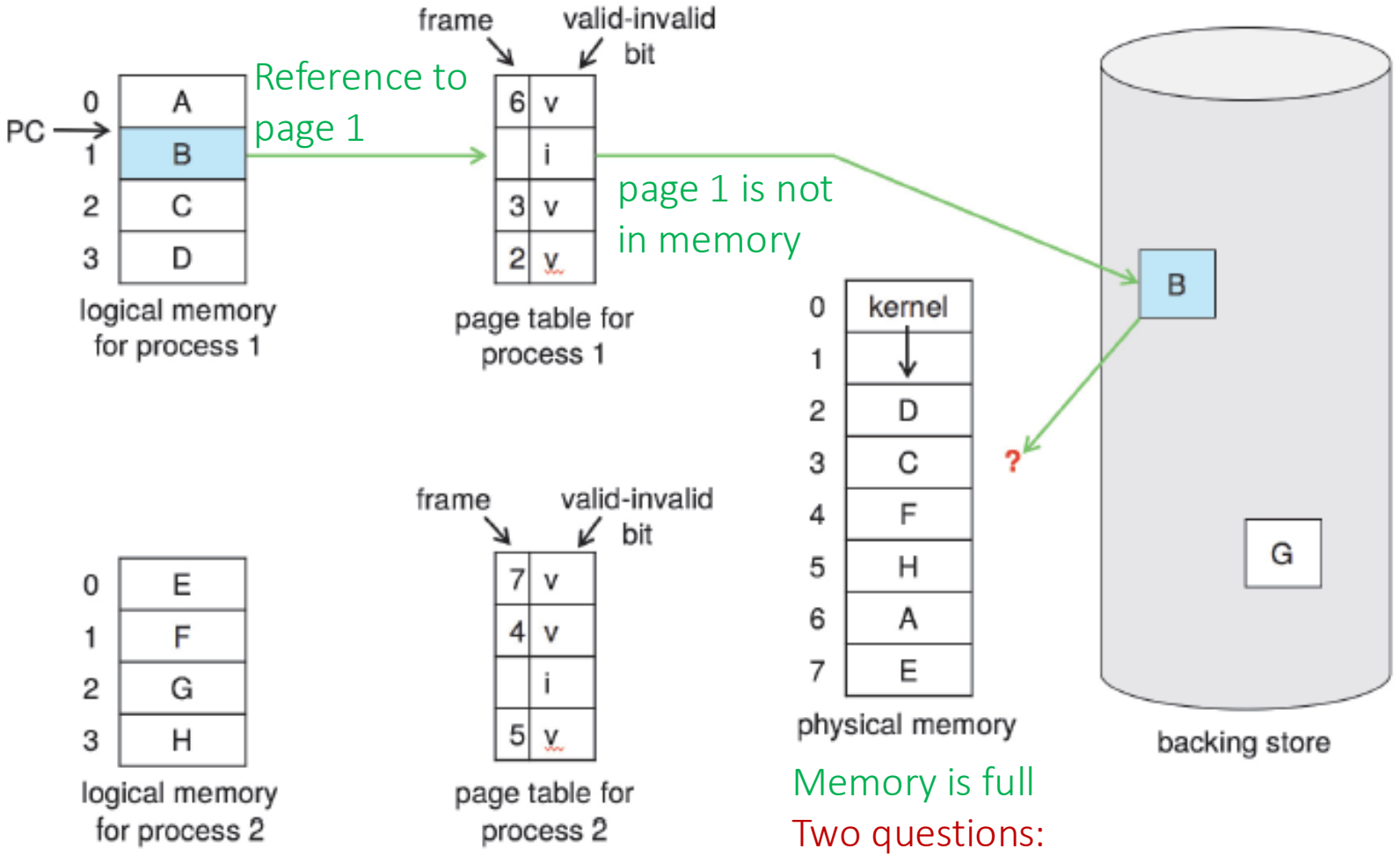




What Happens if There is no Free Frame?

- When there is a page fault, a new page should be loaded in
  - Find a free frame for this new page to load in
  - Question: Memory has used up, i.e., no frame is available for a new page
  - Solution: swap pages out from memory
- Page replacement
  - Find some page in memory, but not really in use, page it out
  - Algorithm consideration
    - How to know which frames are free?
    - If no frame is free, which frames should be freed (i.e., which pages should be replaced, a victim)
      - expectation: minimum number of page faults
- Same pages may be brought into memory several times

Need For Page Replacement



Memory is full

Two questions:

1. Replace page from p1 or p2?
2. Which page to replace?

Global vs. Local Allocation

- Question 1: from which process a page is replaced?
  - Global allocation
    - Process selects a replacement frame from the set of all frames
    - One process can take a frame from another process.
    - The # of frames allocated to each process will change.
  - Local allocation
    - Each process selects from only its own set of allocated frames.
    - The # of frames allocated to each process does not change.

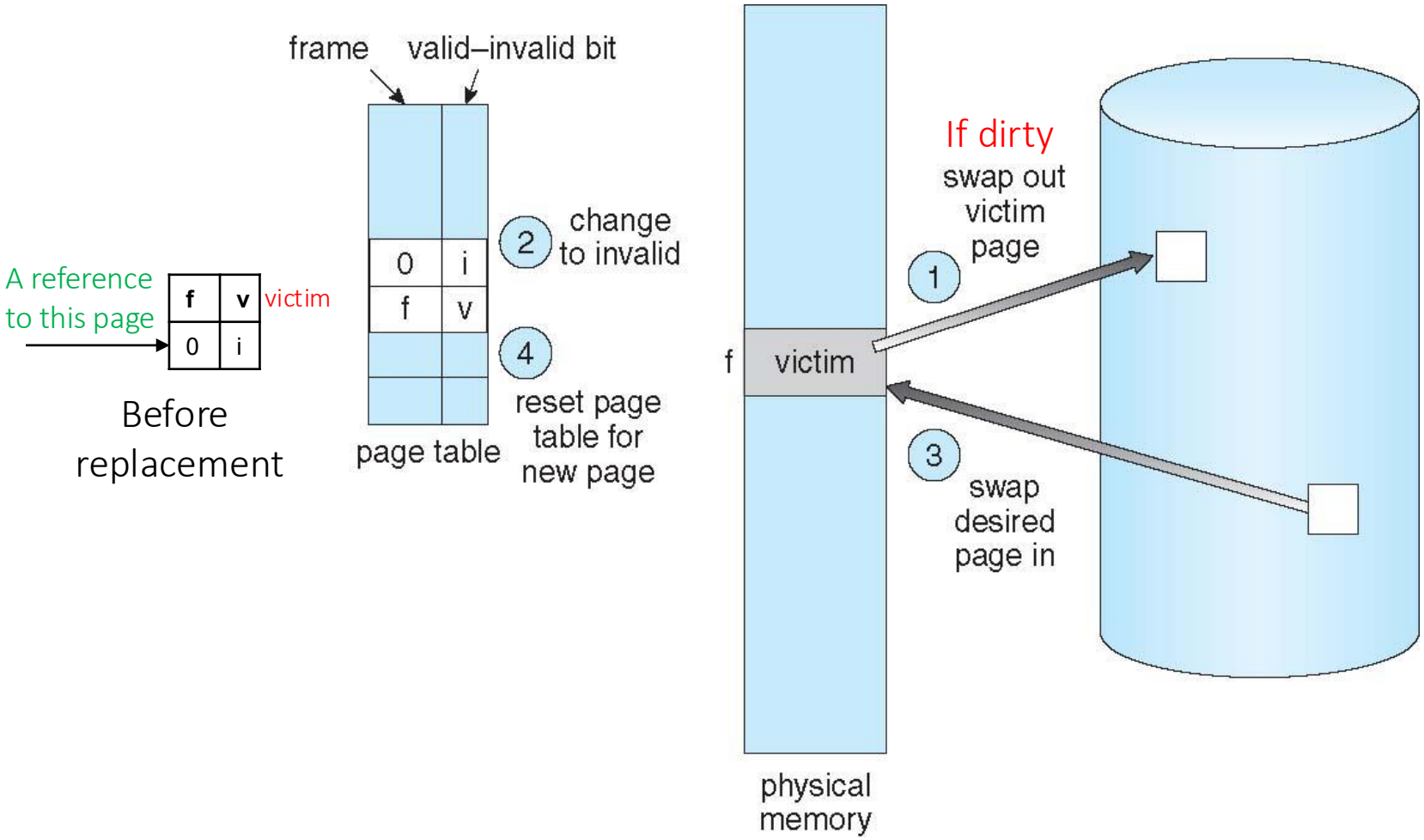
Global vs. Local Allocation

- Which is better: Global or Local
  - Global replacement
    - Process execution time can vary greatly
    - But greater throughput, so more commonly used
  - Local replacement
    - More consistent per-process performance
    - But possibly under-utilized memory

Page Replacement

- Question 2: Which pages to replace (be removed from RAM)?
  - Two kinds of pages are good candidates
    - pages which are not used by a process right now
      - If a page has not been used for a while, maybe it will not be used in near future
    - Pages that are not modified (called dirty) recently
      - These kinds of pages do not need to be paged out
  - An algorithm is needed to find the right frame to free
    - which pages have not been used recently?
    - which pages have not been modified recently?
    - "recent": how to decide the period for "recent"?
  - Victim page
    - The page which will be replaced

Page Replacement



Page and Frame Replacement Algorithms

- Frame-allocation algorithm determines
  - How many frames to give each process
- Page-replacement algorithm
  - Which frames to replace
  - Want lowest page-fault rate on both first access and re-access
- Algorithm evaluation
  - Run it on a particular string of memory references (reference string) and compute the number of page faults on that string
  - String is just page numbers, not full addresses
  - Results depend on number of frames available

Page Replacement Algorithms

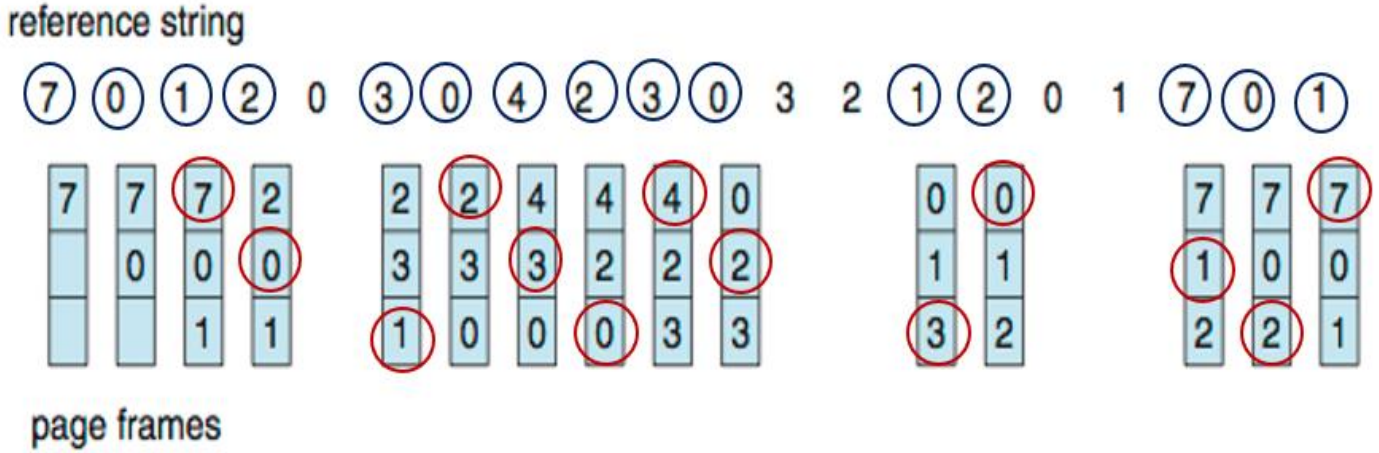
- Algorithms
  - FIFO
  - Optimal
  - Least Recently Used (LRU)
  - Second chance
  - Enhanced second chance
- Reference string used in the examples
  - 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
  - Each number is the page number to reference

FIFO

- Rule
  - Replace the oldest one
- Implementation
  - OS maintains a circular queue of all pages
    - Page at head of the list: Oldest one
    - Page at the tail: Recent arrival
  - When there is a page fault
    - Page at the head is removed
    - New page added to the tail

First-In-First-Out (FIFO) Algorithm

- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (i.e., 3 pages can be in memory at a time)

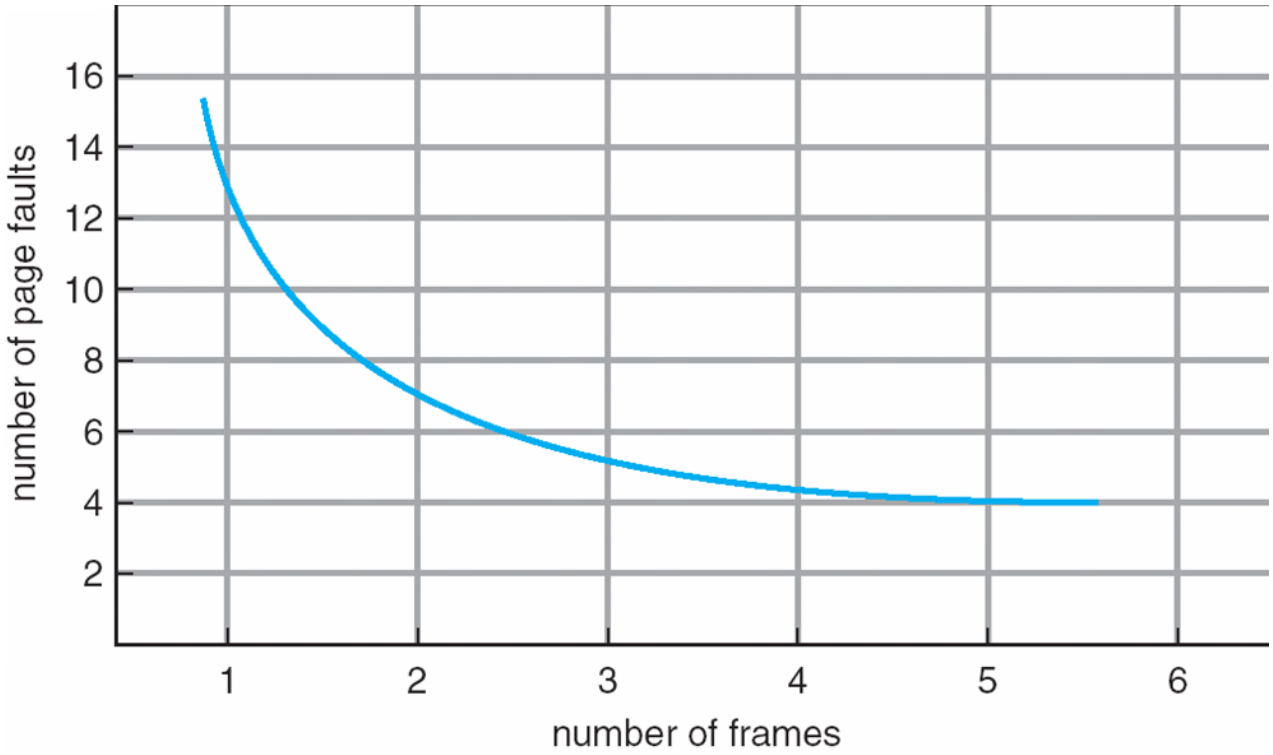


Total page faults: 15

Thinking:

Can adding more frames reduce the number of page faults?

Graph of Page Faults Versus The Number of Frames



Intuition but not Reality

FIFO Illustrating Belady's Anomaly

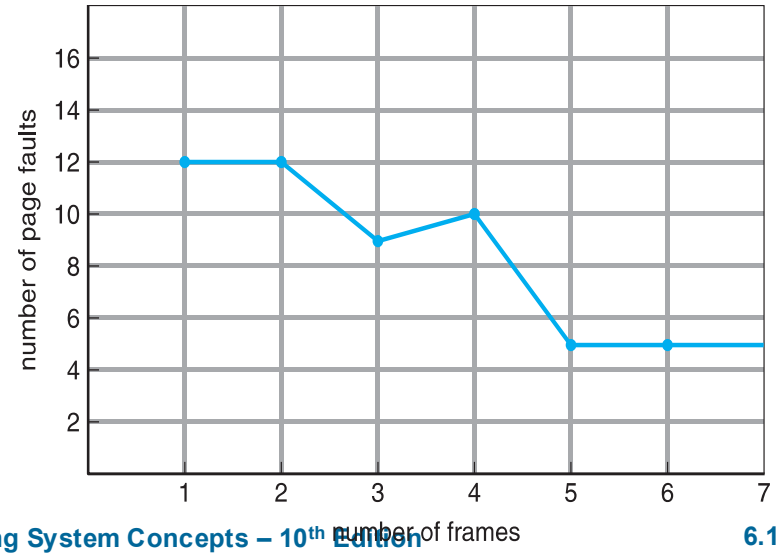
Consider 1,2,3,4,1,2,5,1,2,3,4,5

Adding more frames might cause more page faults!

1	2	3	4	1	2	5	1	2	3	4	5	page fault
1	1	1	4	4	4	5	5	5	5	5	5	
	2	2	2	1	1	1	1	1	3	3	3	9
		3	3	3	2	2	2	2	2	4	4	
1	1	1	1	1	1	5	5	5	5	4	4	
	2	2	2	2	2	2	1	1	1	1	5	10
		3	3	3	3	3	3	2	2	2	2	
			4	4	4	4	4	4	3	3	3	

3 frames

4 frames

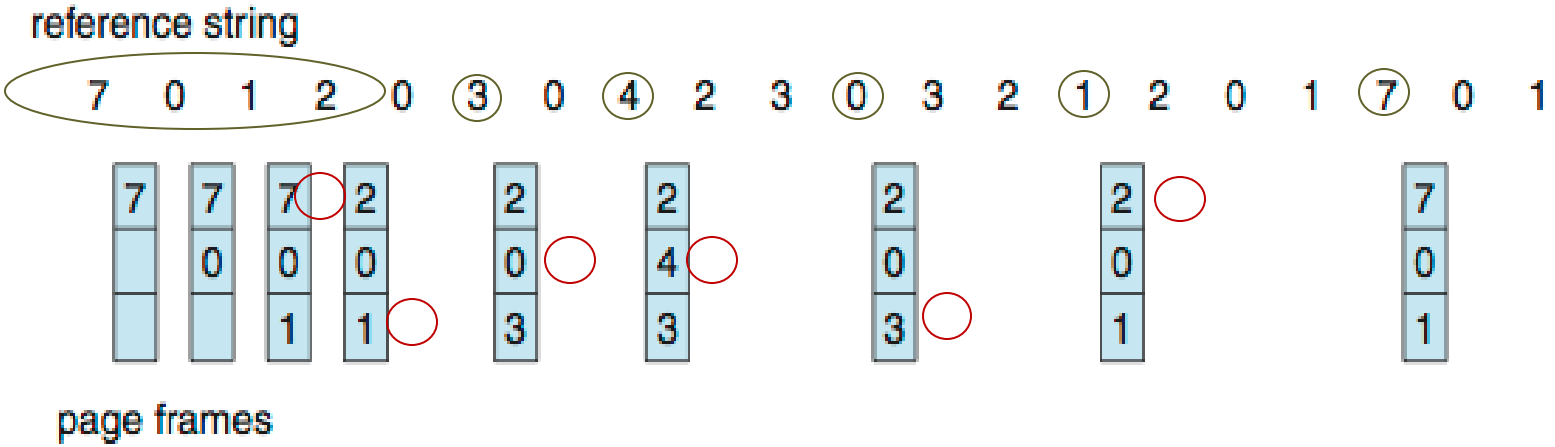


Belady's **Anomaly**

For some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases.

Optimal Algorithm

- Replace the page that will not be used for longest period of time

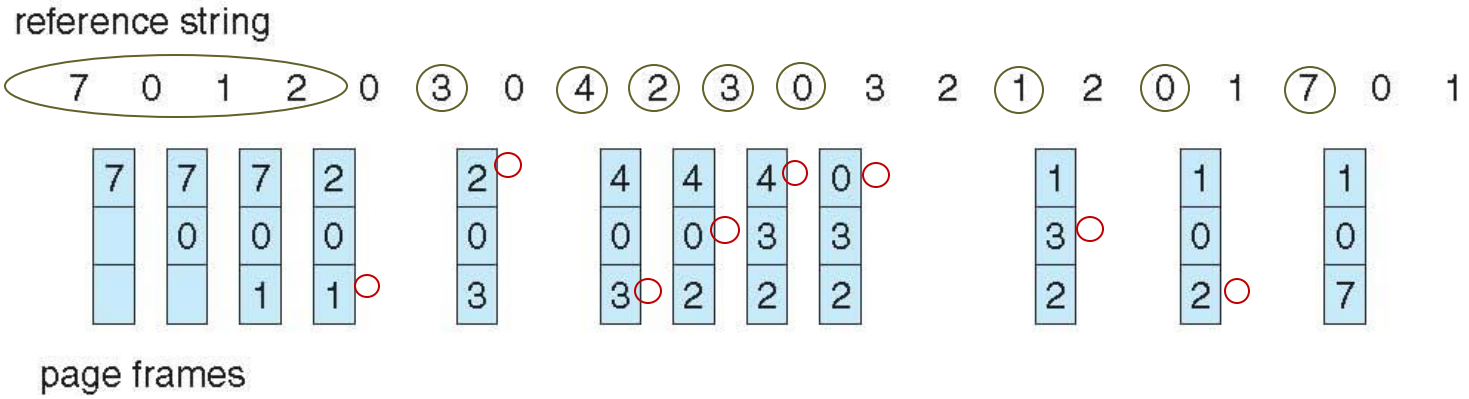


Total page faults: 9

- Problem
  - Can't read the future
- This method can only be used to measure how other algorithms are close to the optimal

Least Recently Used (LRU) Algorithm

- Replace page that has not been used in the most amount of time

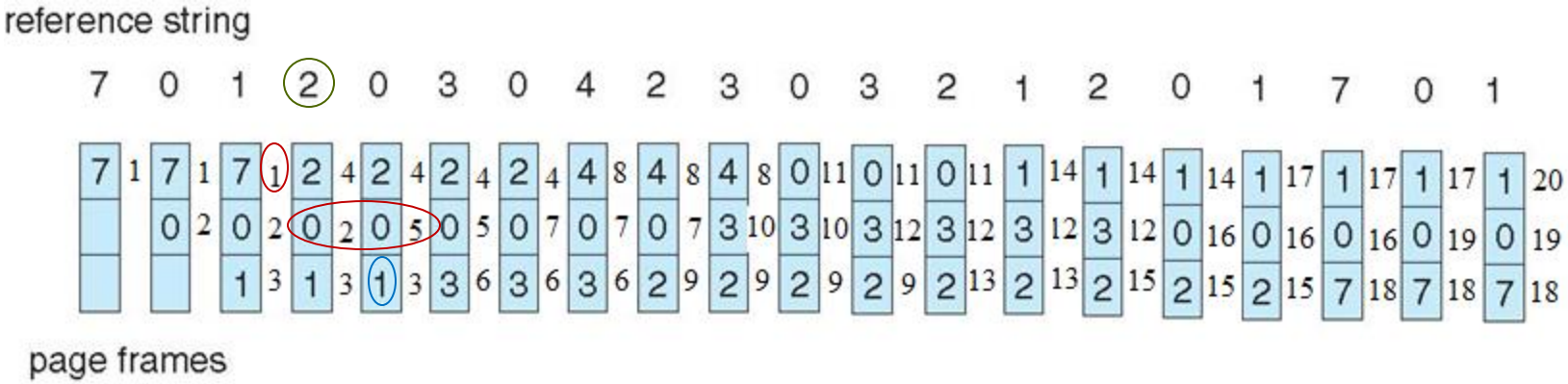


Total page faults: 12

- Better than FIFO(15) but worse than OPT(9)
- LRU is a good algorithm and frequently used
- LRU and OPT don't have Belady's Anomaly

LRU Algorithm Implementation 1: Counter

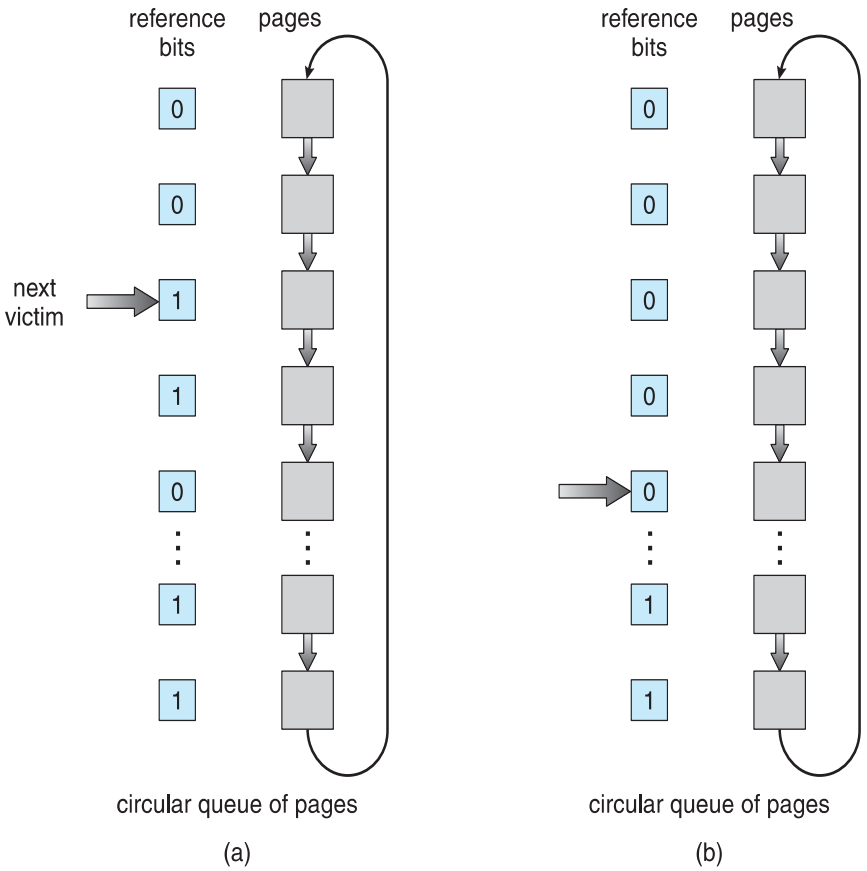
- Counter implementation
  - There is a global counter that increases by 1 whenever a page is referenced. Every page in the memory has its own counter
  - When a page is referenced, its counter is synchronized with the global counter
  - When a page needs to be replaced, find the page in the memory with the smallest value



Thinking:

What is the disadvantage of this algorithm (time issue)?

LRU Approximation Algorithms



- LRU approximation algorithm 2
  - Second-chance algorithm
    - Use a circular FIFO and reference bit for each page in memory
    - If page to be replaced has
      - reference bit = 0
        - » replace it
      - reference bit = 1
        - » set reference bit to 0
        - » search for the next page

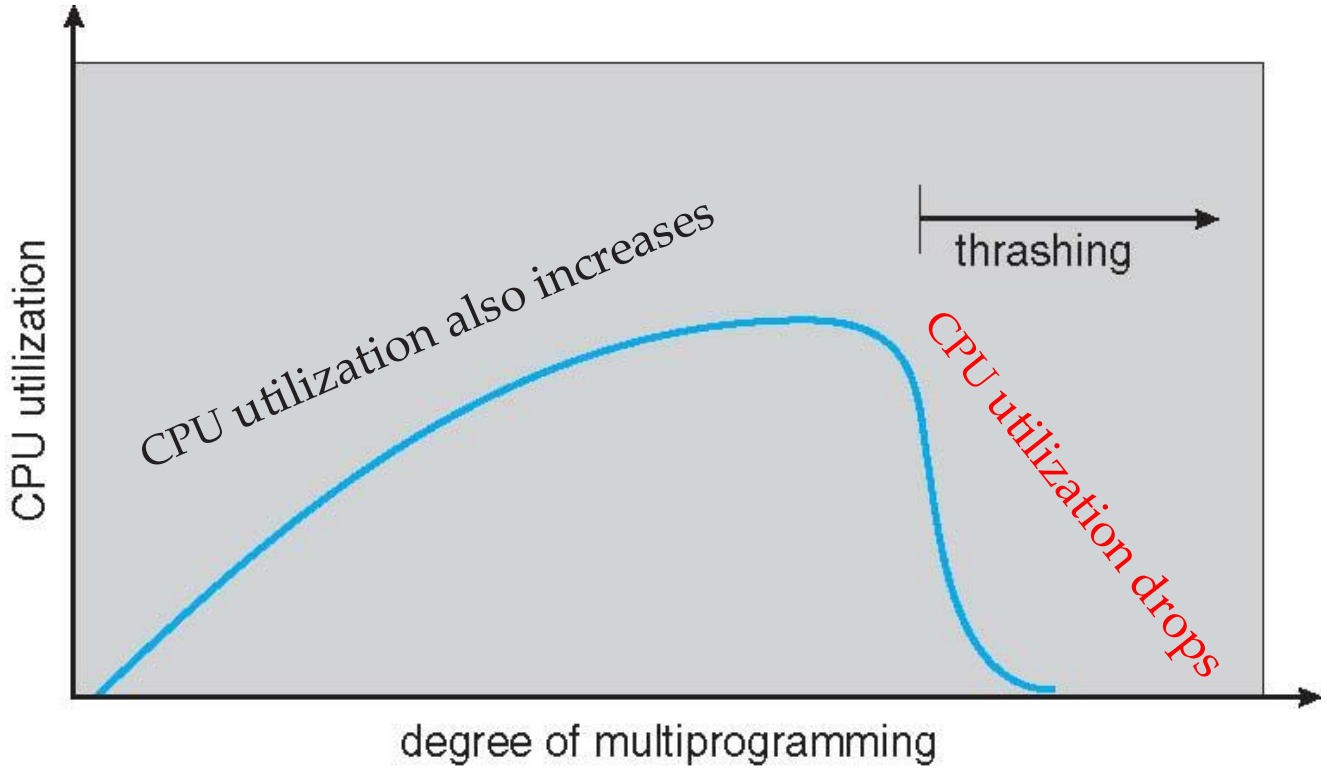
Allocation of Frames

- Two major allocation schemes
  1. Fixed allocation
     - Equal allocation
       - Each process is allocated same number of frames
       - Disadvantage
         - » Space waste for small process
     - Propositional allocation
       - Allocate frames according to the size of a process
       - Disadvantage
         - » Process size might be changed during the execution
  2. Priority allocation
     - the ratio of frames depends on the combination of size and priority of a process
     - Replace the pages of process with lower priority

Thrashing(系统颠簸)

- If a process does not have “enough” frames in memory, the page-fault rate is very high
  - Replace page frequently
  - The replaced page might be used again
  - This leads to:
    - Low CPU utilization
    - Operating system keeps adding new process, trying to increase CPU utilization
    - Things get worse -> entering a bad cycle
    - Thrashing
      - A process is busy swapping pages in and out, instead of doing useful work.

Thrashing (Cont.)

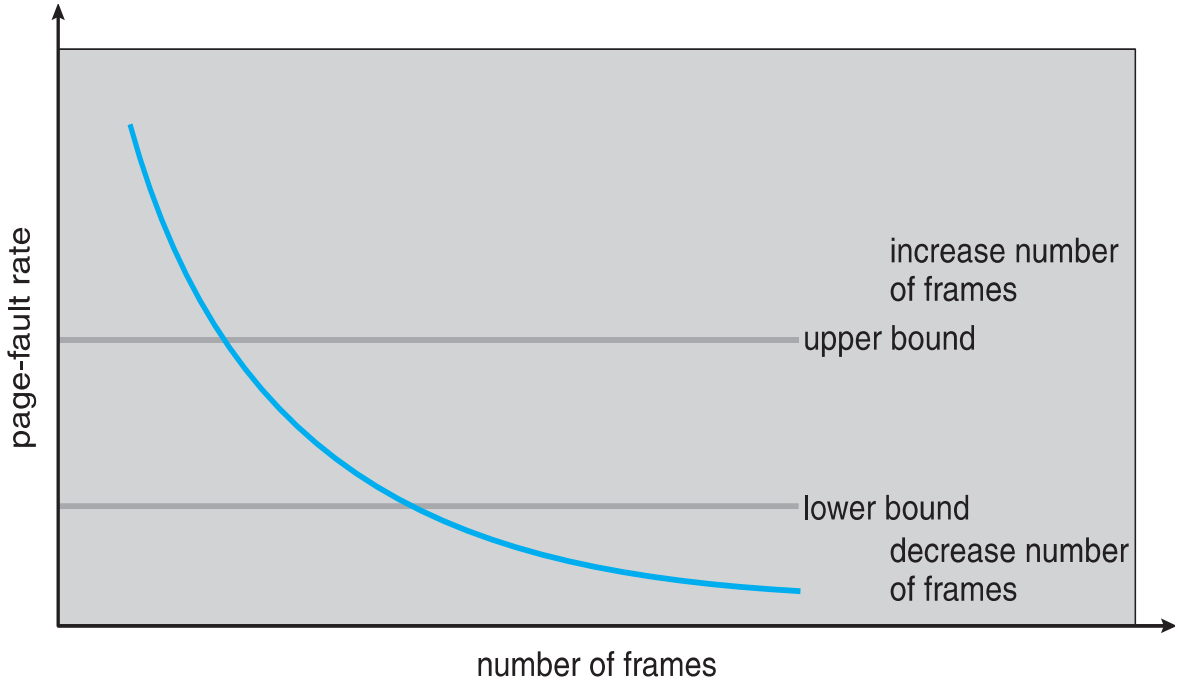


Solutions to thrashing

1. Decrease the degree of multiprogramming
2. Establish "acceptable" page-fault frequency (PFF) rate and use local replacement policy
3. Install enough physical memory (hardware)
4. Install faster hard disk

Page-Fault Frequency

- If actual rate too low, process loses frame (we think too many frames assigned, remove some frames)
- If actual rate too high, process gains frame



Chapter 13: File-System Interface

Outline

- File Concept
- Access Methods
- Disk and Directory Structure
- File-System Mounting
- File Sharing
- Protection

Objectives

- To explain the function of file systems
- To describe the interfaces to file systems
- To discuss file-system design tradeoffs, including access methods, file sharing, file locking, and directory structures
- To explore file-system protection

File Concept

- A file is
  - a named collection of related information that is recorded on secondary storage (e.g. Disk, SSD, Flash, etc.)
  - a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator
  - Smallest logical storage unit for user view
- A file
  - uses contiguous logical address space
  - represents programs or data
- File system resides on secondary storage (disks)
  - Provides user interface to storage, mapping logical to physical
  - Provides efficient and convenient access to disk by allowing data to be stored, located, retrieved easily

File Attributes

- A file usually has following basic attributes
  - Name – only information kept in human-readable form
  - Identifier – unique tag (number) identifies file within file system
  - Type – needed for systems that support different types
  - Location – pointer to file location on device
  - Size – current file size
  - Protection – controls who can do reading, writing, executing
  - Time, date, and user identification – data for protection, security, and usage monitoring
- Information about files（metadata）are kept in the directory structure, which is maintained on the disk

File Operations

- File is an abstract data type.
- Six basic file operations
  1. Create a file
  2. Write a file
  3. Read a file
  4. Reposition within a file
  5. Truncate a file
     - File is reset to length zero
     - File space is released
     - File attributes are unchanged
  6. Delete a file
- Except file creation and deletion, all other operations need to open the file first and then close the file after the operation

File Types – Name, Extension

- The OS uses the extension to indicate the type of the file
- Most common types
  - .c
  - .exe
  - .docx
  - .pdf
  - .jpg
  - .png
  - .zip

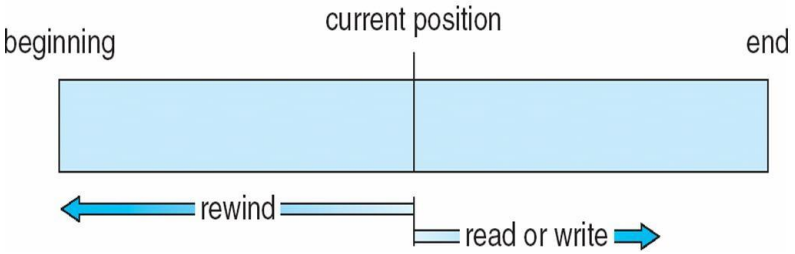
file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Common file types

File Access Methods

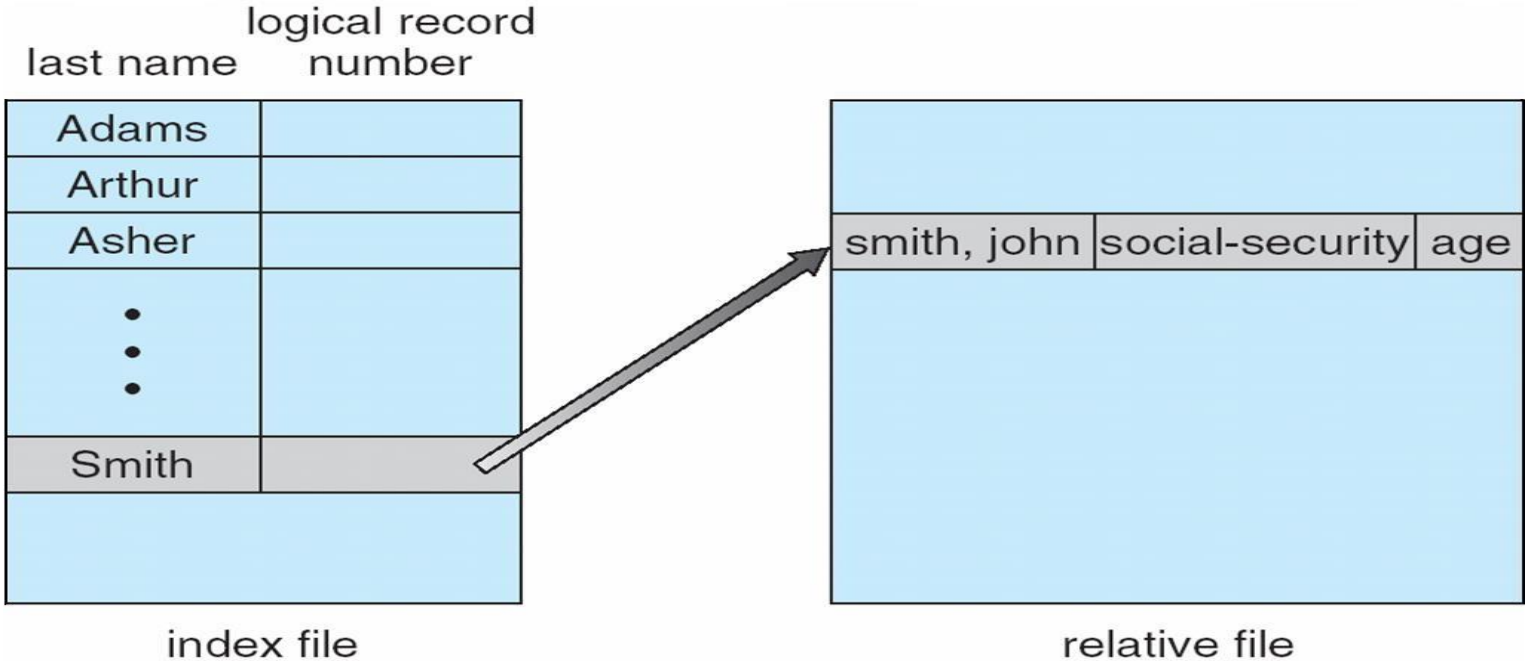
- Sequential Access
  - Commands
    - read_next
    - write_next
    - reset (to the beginning)
- Direct/Random Access
  - Commands
    - read next
    - write next
    - read *n*
    - write *n*
    - position to *n*
    - rewrite *n*

  (*n* = an index relative to the beginning of the file)



sequential access	implementation for direct access
reset	*cp* = 0;
read next	*read cp*; *cp* = *cp* + 1;
write next	*write cp*; *cp* = *cp* + 1;

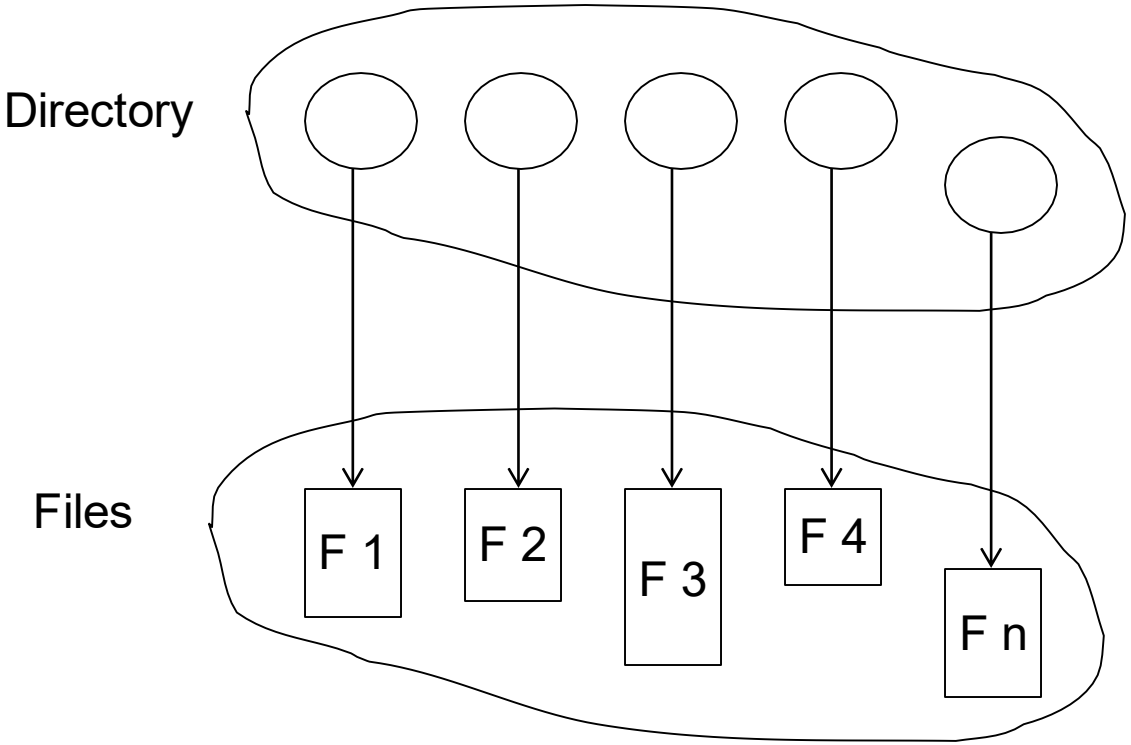
Other Access Method: Index and Relative Files



- Keep index in memory for fast determination of location of data to be operated on
  - E.g., VMS OS

Directory Structure

- A directory structure (per file system) is used to organize the files

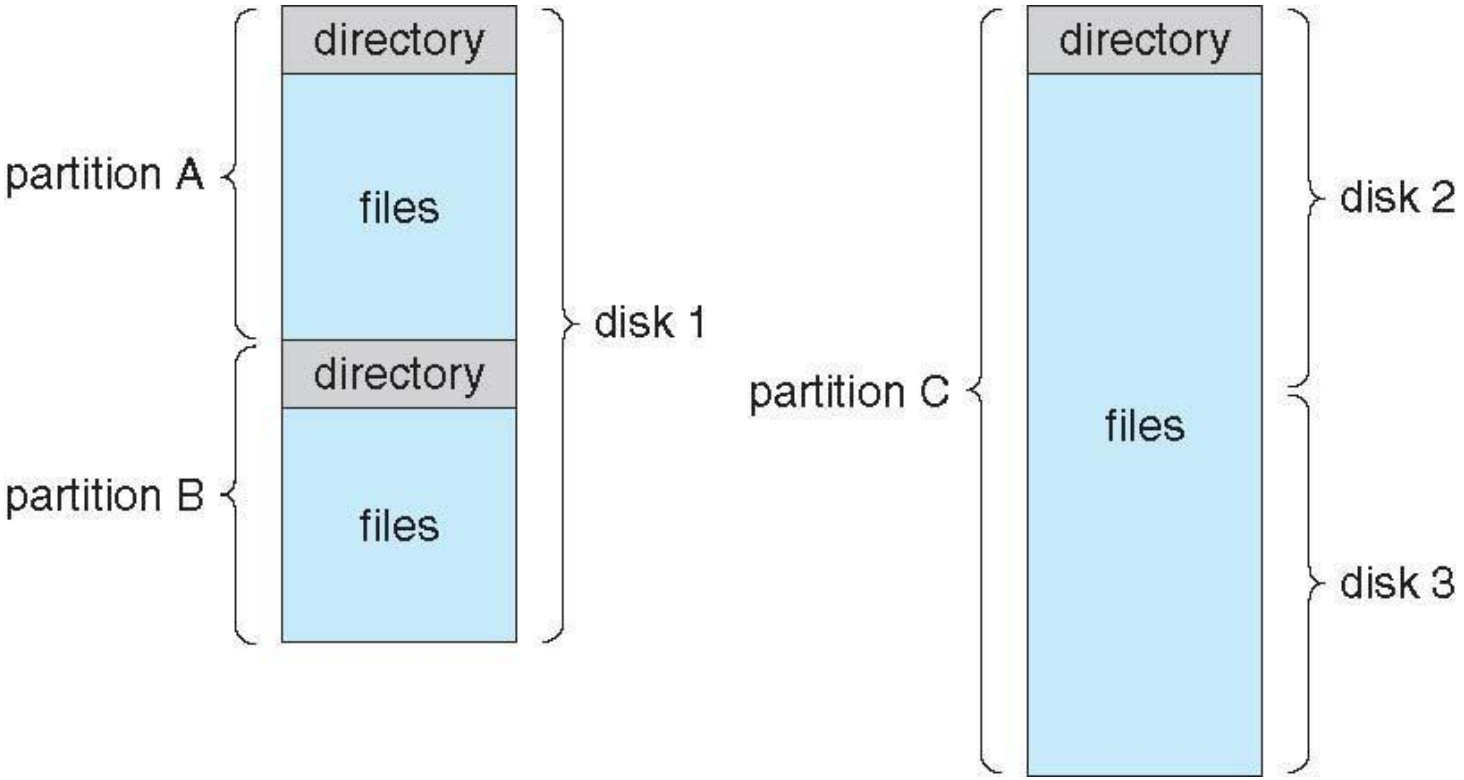


Both the directory structure and the files reside on disk

Disk Structure

- A drive is a physical block disk.
- A drive can be subdivided into several logic block disks (partitions)
- Disk or partition can be used raw, i.e., without a file system
- Partition can be formatted with a file system (cooked)
  - Root partition contains the OS kernel, and other system files
  - Other partitions can hold other OSes, other file systems, or be raw.
- A volume is a single accessible storage area with a single file system, a volume can contain multiple partitions
- File system types
  - General-purpose file systems
    - Mostly used, like Windows
  - Special-purpose file systems
    - E.g., Solaris's contract file system

A Typical File-system Organization



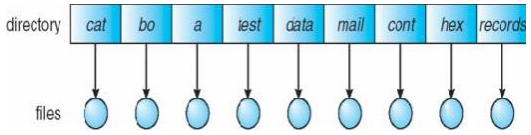
A typical file-system organization

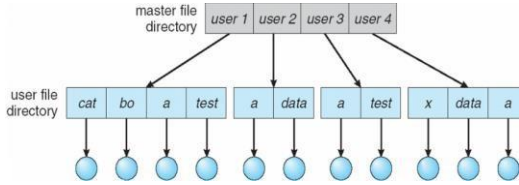
Operations Performed on Directory

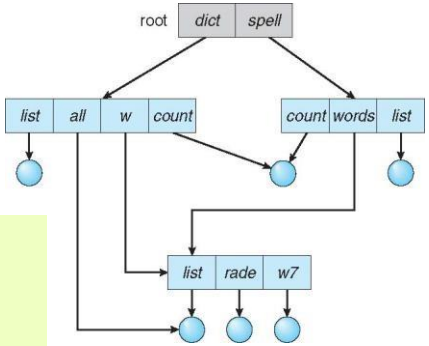
- Basic operations
  - Search for a file
  - Create a file
  - Delete a file
  - List a directory
  - Rename a file
  - Traverse the file system

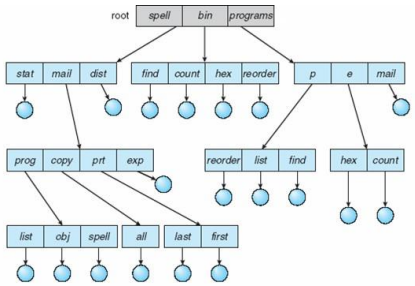
Operations Performed on Directory

- The directory is organized logically to obtain
  - Efficient location of a file
  - Naming convenient to users
    - Two files in different directories can have the same name
    - The same file can have several different names
  - Grouping of files with similar properties
    - E.g., put photos in a directory
- Directory organization
  - Single level directory
  - Two-level directory
  - Tree-structured directories
  - Acyclic graph directories







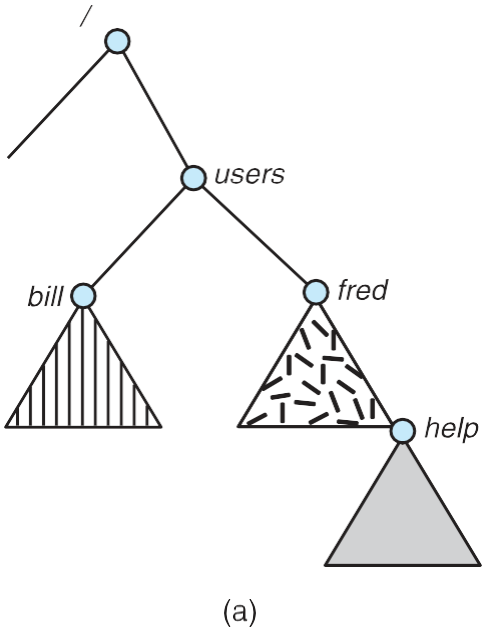


Thinking:

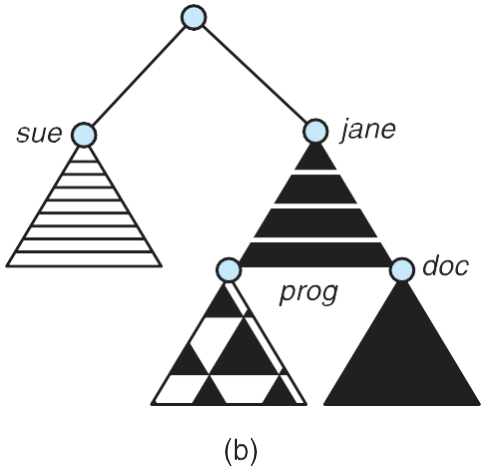
1. Can you tell the organizations of these four examples?
2. Disadvantages for single-level and two-level?

File System Mounting

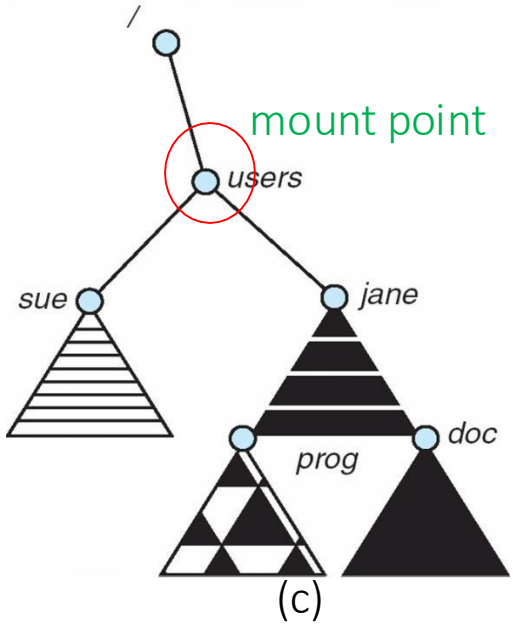
- A file system must be mounted before it can be accessed
- An unmounted file system can be mounted at a mount point
- The amount point must be an empty directory, i.e., original file system at the mount point must be removed if exists



Existing system



Unmounted volume



Mounted volume

Chapter 14: File System Implementation

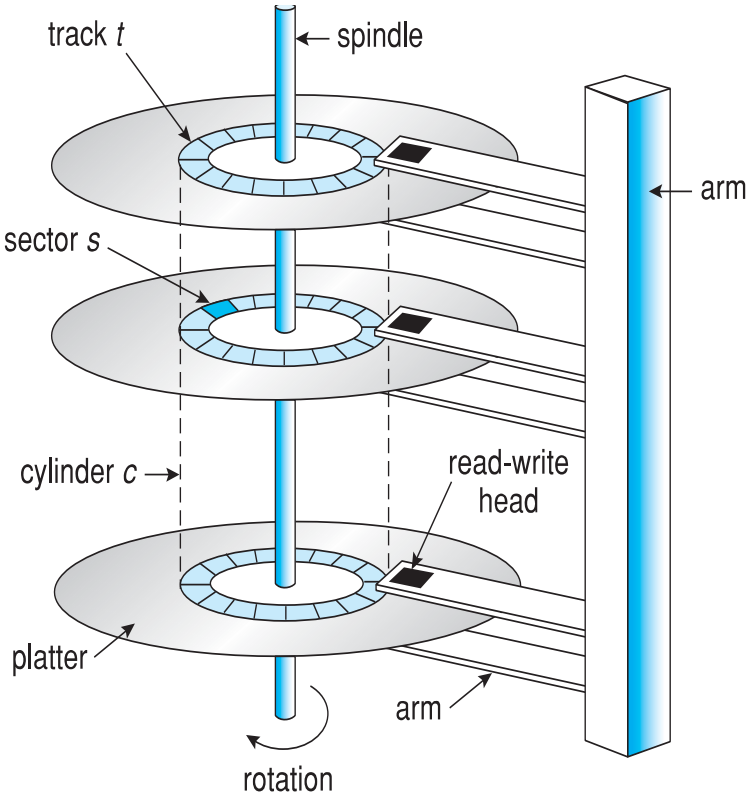
Outline

- File-System Structure
- File-System Operations
- Directory Implementation
- Allocation Methods
- Free-Space Management
- Efficiency and Performance
- Example: WAFL File System

Objectives

- Describe the details of implementing local file systems and directory structures
- Discuss block allocation and free-block algorithms and trade-offs
- Explore file system efficiency and performance issues
- Describe the WAFL file system as a concrete example

Moving-head Disk Mechanism



- One sector is one block
- Disk systems typically have a well-defined block size (usually 512 bytes) determined by the size of a sector
- Disk provides in-place rewrite and random access
- All disk I/O is performed in units of one block (physical record) of the same size, indexed by block no. in a linear array
  - E.g., I/O sequence 20, 35, 11
- It is unlikely that the physical record size will exactly match the length of the desired logical record
  - Logical records may even vary in length
  - Packing a number of logical records into physical blocks is a common solution to this problem

OS and File System

- One operating system can support one or more file systems
- Many operating systems allow administrators or users to add file systems
- Each file system has its own format
  - disk file-system formats
  - CD-ROM and DVD file-system formats (ISO 9660)
- Unix supports
  - Unix File System (UFS), which is based on Fast File System (FFS)
- Linux supports over 130 files systems
  - Extended File System (Standard Linux file system)
    - Versions: ext2, ext3, ext4
- Windows supports
  - Disk file formats
    - 16 bit File Allocation Table (FAT),
    - 32 bit File Allocation Table (FAT32),
    - New Technology File System (NTFS)
  - CD, DVD Blu-ray file system formats

File-System Implementation

- Structures used in file system implementation
  - Two categories of structures
    - on-disk (on-storage) structures contain information about
      - how to boot an operating system stored in disk,
      - the total number of blocks,
      - the number and location of free blocks,
      - the directory structure, and
      - individual files
    - in-memory structures contain information used for
      - file-system management
      - performance improvement via caching

On-disk (On-storage) Structures

Volume control block

- Volume details
  - Total # of blocks, free blocks,
  - block size,
  - free-block count and free-block pointers,
  - free-FCB count and FCB pointers
- Ext (Linux): called superblock
- NTFS (Windows): called master file table

Directory structure

- Organize the files
- UFS (Unix): includes file names and associated inode (same as File Control Blocks) numbers
- NTFS(Windows) : File Control Blocks (FCB), called master file table

A volume: logical disk drive, perhaps a partition

Boot control block

- First block of volume
- Information for booting OS if OS is in this volume; otherwise empty
- UFS(Unix): called boot block
- NTFS(Windows): called partition boot sector

Each file has a File Control Block or inode

- Unix: Indexed using inode number, includes permissions, size, dates

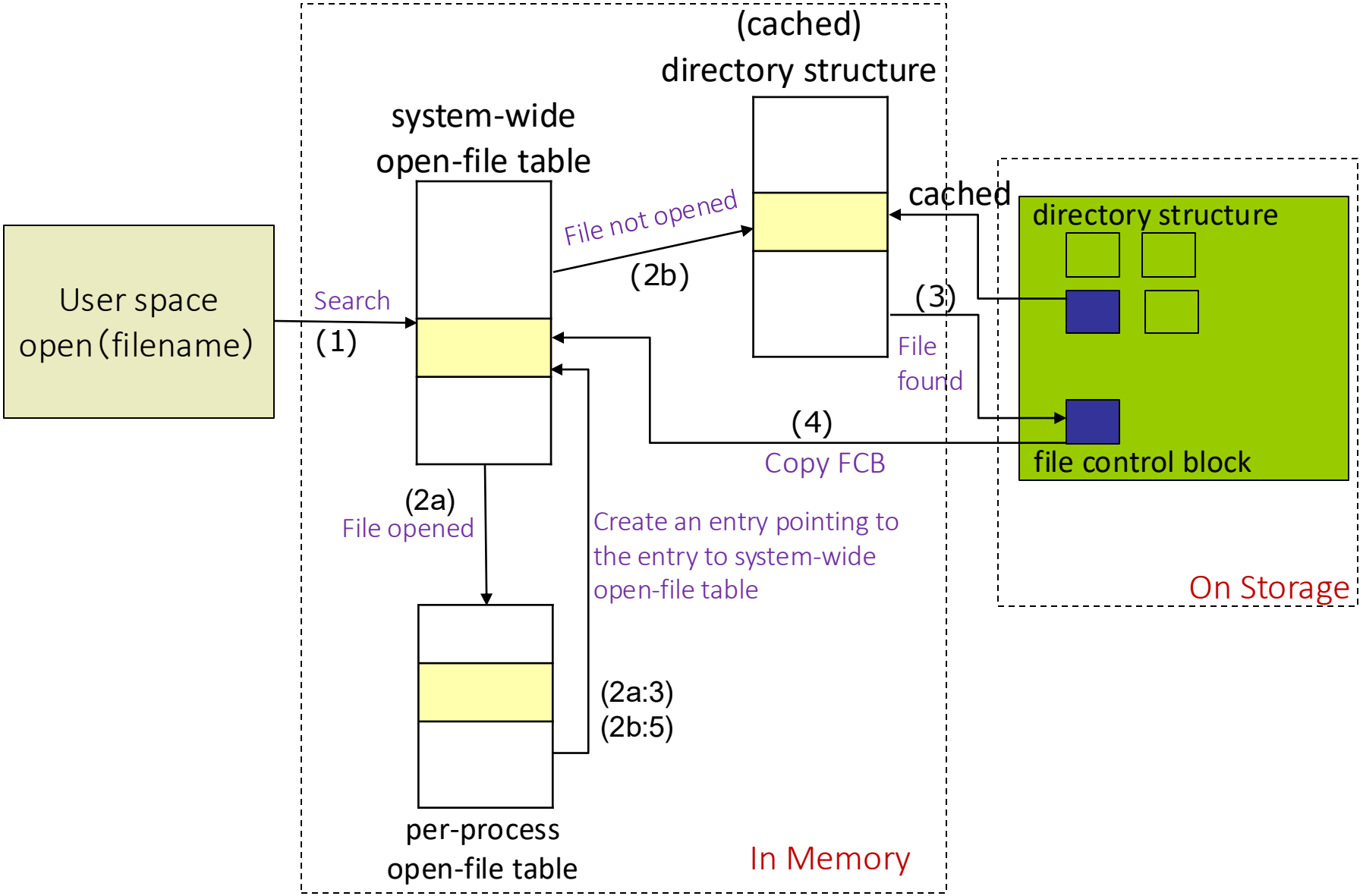
In-Memory Structures

- Mount table
  - stores file system mounts, mount points, file system types
- Directory-structure cache
  - holds the directory information of recently accessed directories
- system-wide open-file table
  - for all the processes
  - contains a copy of the FCB of each file
  - tracks all open files
  - keeps the counter of processes that have opened each file
  - contains process-independent information
    - E.g., the location of the file on disk, access dates, and file size
- per-process open-file table
  - contains pointers to appropriate entries in system-wide open-file table
  - tracks all the files that a process has opened
  - keeps a file pointer that indicates the last read-write position of the file for each opened file
  - keeps file access rights

File Open and Close

- Open a file
  - Search the directory structure on disk for the file and copy the content of entry（metadata）to system-wide open file table if the file is opened for the first time
  - Update the per-process open-file table by adding a pointer to system-wide open file table
- Close a file
  - Remove the file entry in system-wide open file table if no process to use it any more
  - Update the per-process table by removing a pointer to system-wide open file table

Open A File

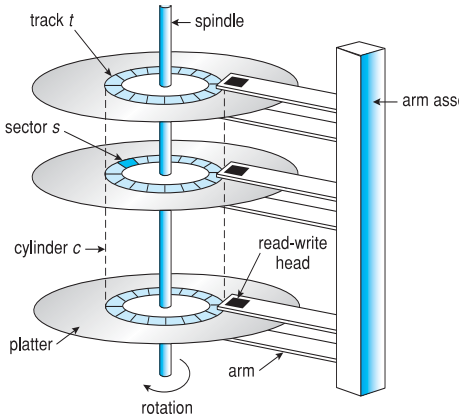

(cached)
directory structure
system-wide
open-file table
cached
directory structure
File not opened
(2b)
User space
open(filename)
Search
(1)
(3)
File
found
(4)
Copy FCB
file control block
(2a)
File opened
Create an entry pointing to
the entry to system-wide
open-file table
On Storage
(2a:3)
(2b:5)
per-process
open-file table
In Memory

Allocation Methods

- How disk blocks are allocated for files
  - Contiguous allocation(not common now)
  - Linked allocation (e.g. FAT32 in Windows)
  - Indexed allocation (e.g. ex3 in Unix)

Allocation Methods - Contiguous

- Contiguous allocation
  - Each file occupies set of contiguous blocks
  - Each file can be located by start block # and length (total number of blocks)
    - E.g., A file that occupies n blocks: b, b+1, b+2, b+3, …, b+n-1
  - Advantages
    - Simple
    - Each to access
    - Minimal disk head movement
      - Normally on same track, or at most move to next track.
  - Problems
    - Not suitable for a file with varied size
      - file size must be determined at the beginning (size-declaration)
    - External fragmentation
      - Solution: compacts all free space into one contiguous space
      - Compaction can be done off-line (down time) or on-line
    - Internal fragmentation



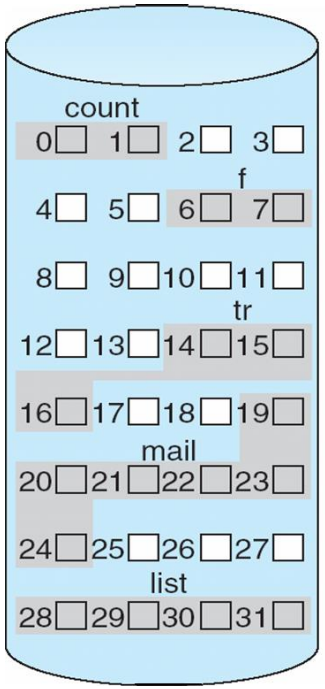
Contiguous Allocation: An Example

- Mapping from logical to physical
- Assumption
  - LA: Logical Address, starting from 0
  - Block size: 512 bytes
- Physical address
  - (Q + File physical start block) * 512 + R
    - Q: block no. relative to start
    - R: displacement into the block

Quotient
Q

LA/512

R
Remainder



directory

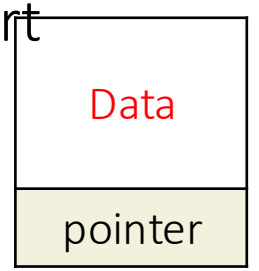
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

File *tr*: 3 blocks in length
Starting at block 14

Thinking:
For file ***tr***, if a byte's LA is 1025, what is its physical address?

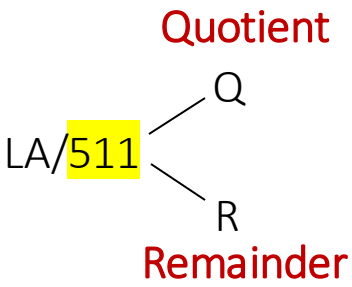
Linked Allocation: An Example

- Each file is a linked list of disk blocks
  - Blocks may be scattered anywhere on the disk
  - Physical address
    - Rth byte in the Qth block in the list (index starts from 0)
      - Q: block index in the list relative to the start
      - R: displacement into the block



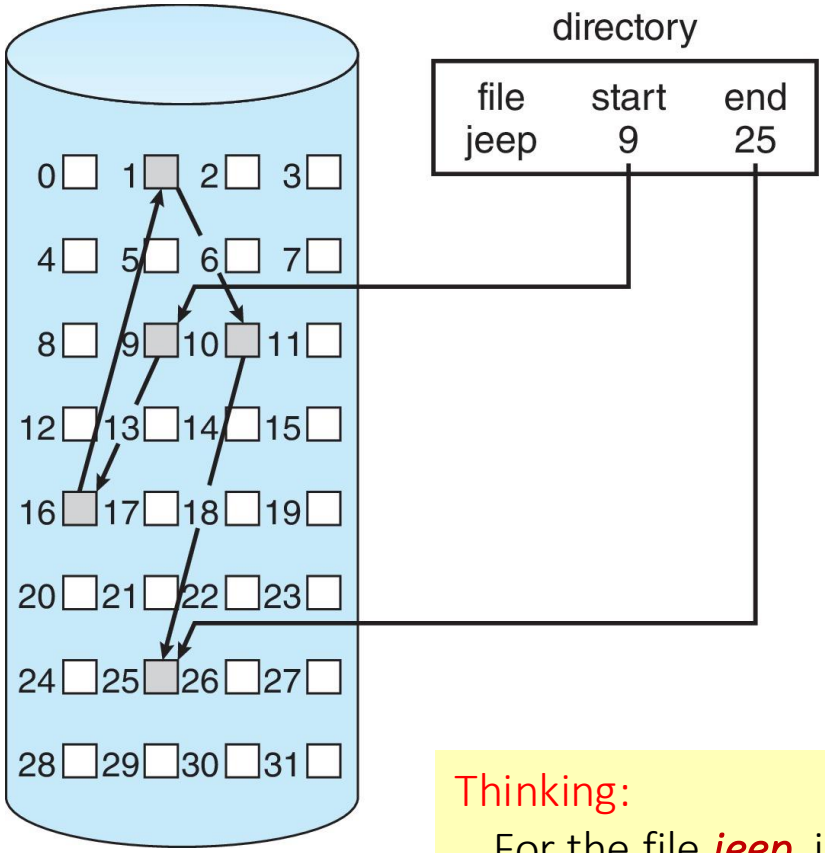
A block

Thinking:
Why divided by 511?



Assumption
Pointer: 1 byte
Block size: 512 bytes

Linked Allocation

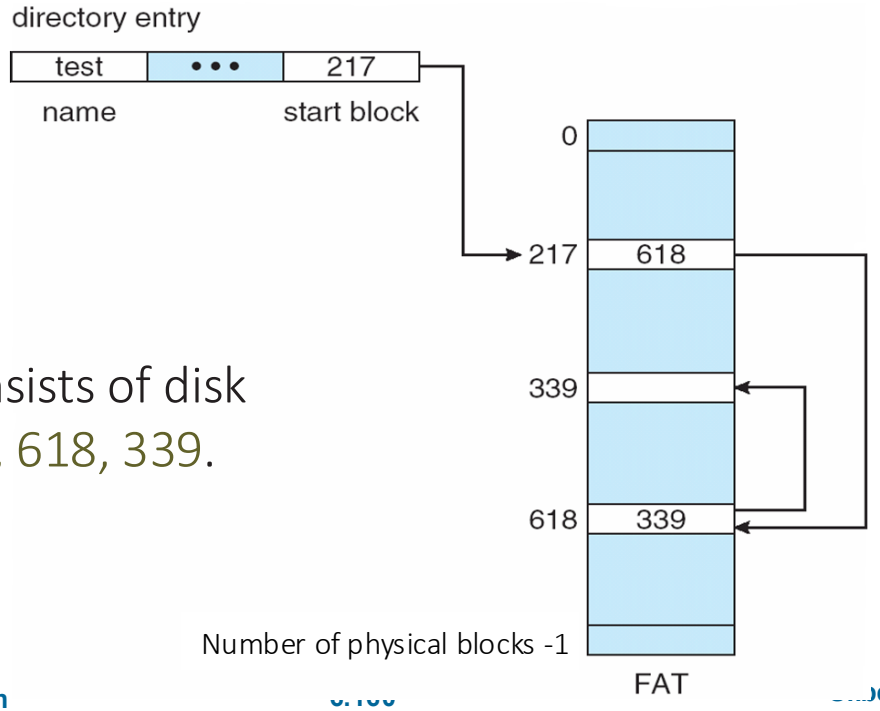


Thinking:

For the file ***jeep***, if LA is 512, what is its physical address?

Allocation Methods – Linked Allocation Variation

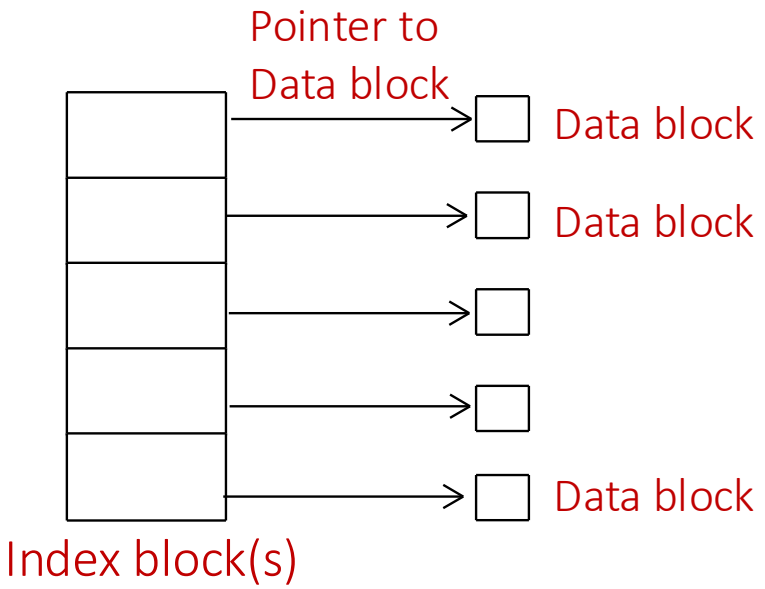
- File Allocation Table (FAT)
  - An important variation on linked allocation used in MS-DOS
  - Indexed in physical block no.
  - Each entry stores the no. of next block for this file
  - Like a linked list, but faster on disk and cacheable
  - Simple and efficient



File *test* consists of disk blocks: 217, 618, 339.

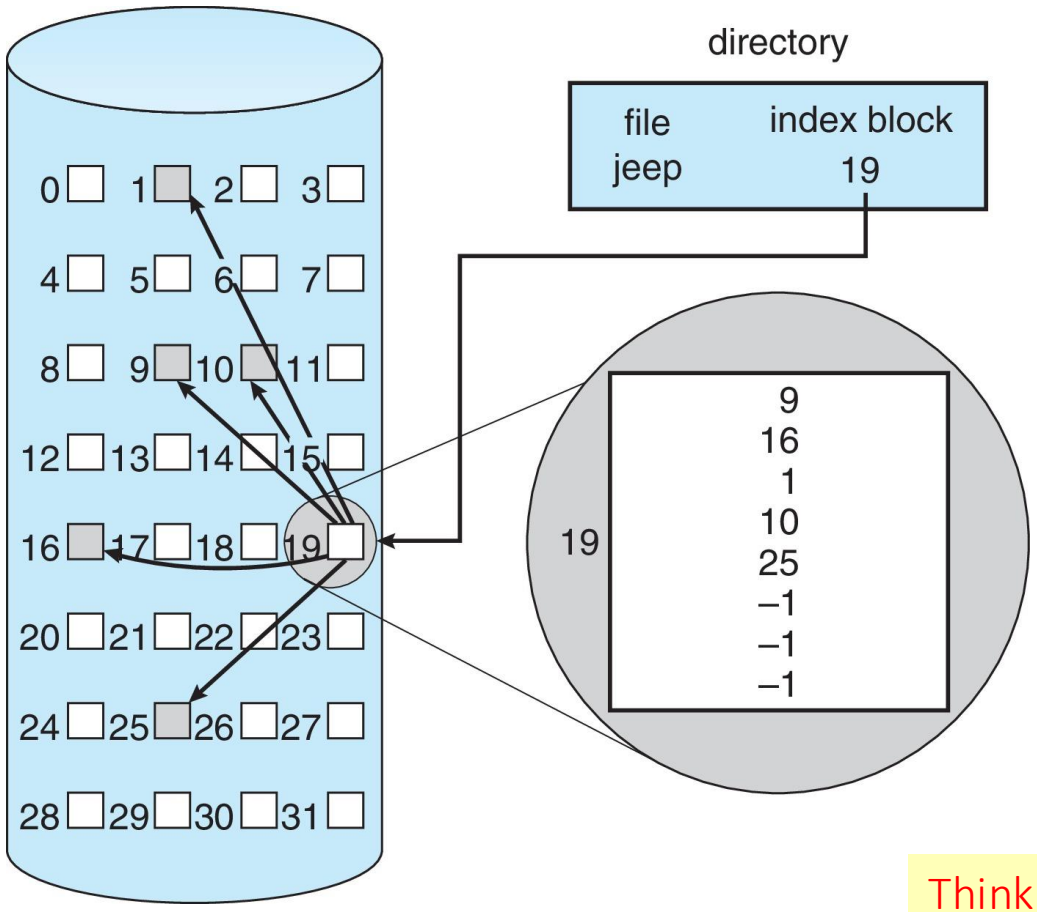
Allocation Methods – Indexed Allocation

- Indexed allocation
  - Bring all the pointers together into the index block(s)
  - Each entry points to a block in a file
  - Each file has its own index block(s) of pointers to its data blocks
  - Advantages
    - Random access
    - No external fragmentation
  - Disadvantages
    - Extra space for index table, more waste space than linked allocation



An index table

Example of Indexed Allocation



Indexed allocation of disk space

Assumption

Pointer: 1 byte

Block size: 512 bytes

Thinking:

1. For the file ***jeep***, if LA is 513, what is its physical address?
2. For a file has size 256K, how many blocks are used as index block?

A Comparison Table

	Contiguous allocation	Linked allocation	Linked Allocation(FAT)	Indexed allocation
allocation	contiguous	scattered	scattered	scattered
Directory hold information	For each file: file name, start block, number of blocks	For each file: file name, start block, end block. Linked through pointer	• For each file: file name, start block • One volume table indexed by block no. including next block no. Linked through block no.	• For each file: Index block no • Index block: Pointers to data blocks
Direct access support	Yes	No	Yes (Partial)	Yes
fragmentation	External, internal	Internal	Internal	Internal
Overhead	No	Pointer	FAT	Index block
File size declaration	Necessary	Not necessary	Not necessary	Not necessary

Indexed Allocation – Schemes in Handling Large Files

- One index block is not enough for large files
- Three schemes to solve this issue
  1. Linked scheme
  2. Multilevel index
  3. Combined scheme

Free-Space Management

- File system need to track available (free) blocks
- Approaches
  1. Bit vector (map)
  2. Linked list
  3. Grouping
  4. Counting

Efficiency and Performance

- The efficient use of storage device space depends on
  - disk allocation method
    - Contiguous, Linked, and Indexed
  - directory structure / algorithms
  - types of data kept in file' s directory entry
  - allocation of metadata structures
  - fixed-size or varying-size data structures
    - e.g. the open file tables
- Ways to improve efficiency and performance
  - Keep data and metadata close together (for hard disk, but not SSD)
  - Use buffer cache for quick access
  - Use asynchronous writes
    - No need to write to disk immediately
  - Use Trim mechanism
    - Keep the unused blocks for writing
    - No need to do garbage collection and erase steps before the device is nearly full