CHAPTER 6: PROCESS SYNCHRONIZATION

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris 2
- Atomic Transactions

Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem (Section 4.4) allows at most $n-1$ items in buffer at the same time.
- Suppose that we modify the producer-consumer code (Section 4.6) by adding a variable *counter*, initialized to 0 and incremented each time a new item is added to the buffer.

  The new scheme is illustrated in the following slide.

- Shared data

```
type item = ... ;
var  buffer: array [0..n–1] of item;
in, out: 0..n–1;
counter: 0..n;
in := 0;
out := 0;
counter := 0;
```

- Producer process

```
repeat
      ...
    produce an item in nextp
      ...
    while counter = n do no-op;
    buffer[in] := nextp;
    in := in+1 mod n;
    counter := counter + 1;
until false;
```

- Consumer process

```
repeat
    while counter = 0 do no-op;
    nextc := buffer[out];
    out := out+1 mod n;
    counter := counter – 1;
        ...
    consume the item in nextc
        ...
until false;
```

- The statements:

  counter := *counter* +1;

  counter := *counter* – 1;

  must be executed *atomically*.

The Critical-Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Structure of process P_i

```
repeat
    entry section
        critical section
    exit section
        remainder section
until false;
```

A solution to the critical-section problem must satisfy the following 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 there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3) **Bounded Waiting**. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

   - Assumption that each process is executing at a nonzero speed.
   - No assumption concerning *relative* speed of the *n* processes.

Trace of initial attempts to solve the problem.

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
repeat
    $\boxed{\textit{entry section}}$
        critical section
    $\boxed{\textit{exit section}}$
        remainder section
until false;
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:
  - **var** *turn*: (0..1);
    initially *turn* = 0
  - *turn* = *i* $\Rightarrow P_i$ can enter its critical section

- Process P_i

```
repeat
    while turn ≠ i do no-op;
        critical section
    turn := j;
        remainder section
until false;
```

- Satisfies mutual exclusion, but not progress.

Algorithm 2

- Shared variables
  - **var** *flag*: **array** [0..1] **of** *boolean*;
    initially *flag*[0] = *flag*[1] = *false*.
  - *flag*[*i*] = *true* $\Rightarrow P_i$ ready to enter its critical section

- Process P_i

```
repeat
    ┌──────────────────────────────┐
    │ flag[i] := true;             │
    │ while flag[j] do no-op;      │
    └──────────────────────────────┘
        critical section
    ┌──────────────────┐
    │ flag[i] := false;│
    └──────────────────┘
        remainder section
until false;
```

- Does not satisfy the mutual exclusion requirement.

Algorithm 3

- Combined shared variables of algorithms 1 and 2.

- Process P_i

```
repeat
    flag[i] := true;
    turn := j;
    while (flag[j] and turn=j) do no-op;
        critical section
    flag[i] := false;
        remainder section
until false;
```

- Meets all three requirements; solves the critical-section problem for two processes.

Bakery Algorithm – Critical section for *n* processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.

- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.

- The numbering scheme always generates numbers in increasing order of enumeration.

  Example: 1,2,3,3,3,3,4,5...

- Notation < ≡ lexicographical order (ticket #, process id #)

  - $(a,b) < (c,d)$ if $a < c$ or if $a = c$ and $b < d$

  - $max(a_0, ..., a_{n-1})$ is a number, k, such that $k \geq a_i$ for $i = 0, ..., n-1$

Bakery Algorithm

- Shared data
  - **var** *choosing*: **array** [0..n–1] **of** *boolean*;
    number: **array** [0..n–1] **of** *integer*;
  - initially *choosing*[i] = *false*, for i = 0,1,..n–1
    number[i] = 0, for i = 0,1,..n–1

```
repeat
    choosing[i] := true;
    number[i] := max(number[0],...,number[n–1])+1;
    choosing[i] := false;
    for j := 0 to n–1
      do begin
        while choosing[j] do no-op;
        while number[j] ≠ 0
            and (number[j],j) < (number[i],i) do no-op;
      end;

      critical section

    number[i] := 0;

      remainder section
until false;
```

Synchronization Hardware

- Test and modify the content of a word atomically.

```
function Test-and-Set (var target: boolean): boolean;
    begin
        Test-and-Set := target;
        target := true;
    end;
```

- Mutual exclusion algorithm
  - Shared data: **var** *lock*: *boolean* (initially *false*)
  - Process P_i

```
repeat

    while Test-and-Set(lock) do no-op;

        critical section

    lock := false;

        remainder section

until false;
```

Semaphore – synchronization tool that does not require busy waiting.

Semaphore *S*

- integer variable
- can only be accessed via two indivisible (atomic) operations

```
wait(S):      S := S – 1;
              if S < 0 then block(S)
signal(S):    S := S + 1;
              if S ≤ 0 then wakeup(S)
```

- *block(S)* – results in suspension of the process invoking it.
- *wakeup(S)* – results in resumption of exactly one process that has invoked *block(S)*.

Example: critical section for *n* processes

- Shared variables
  - **var** *mutex* : *semaphore*
  - initially *mutex* = 1

- Process P_i

```
repeat
    $\boxed{wait(mutex);}$
        critical section
    $\boxed{signal(mutex);}$
        remainder section
until false;
```

Implementation of the *wait* and *signal* operations so that they must execute atomically.

- Uniprocessor environment
  - Inhibits interrupts around the code segment implementing the *wait* and *signal* operations.
- Multiprocessor environment
  - If no special hardware provided, use a correct software solution to the critical-section problem, where the critical sections consist of the *wait* and *signal* operations.
  - Use special hardware if available, i.e., *Test-and-Set*:

Implementation of *wait (S)* operation with the *Test-and-Set* instruction:

- Shared variables
  - **var** *lock* : *boolean*
  - initially *lock* = *false*

- Code for *wait*(S):

```
while Test-and-Set(lock) do no-op;
      S := S - 1;
      if S < 0 then
            begin
                lock := false;
                block(S)
            end
      else lock := false;
```

Race condition exists!

Semaphore can be used as general synchronization tool:

- Execute B in P_j only after A executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
.	.
.	.
.	.
A	*wait*(*flag*)
signal(*flag*)	B

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

  Let S and Q be two semaphores initialized to 1

P_0	P_1
$wait(S)$	$wait(Q)$
$wait(Q)$	$wait(S)$
.	.
.	.
.	.
$signal(S)$	$signal(Q)$
$signal(Q)$	$signal(S)$

- Starvation – indefinite blocking

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

Two types of semaphores:

- *Counting* semaphore – integer value can range over an unrestricted domain.

- *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement.

- Can implement a counting semaphore *S* as a binary semaphore.

  - data structures:

```
var S1: binary-semaphore;
    S2: binary-semaphore;
    S3: binary-semaphore;
    C: integer;
```

  - initialization:

    $S1 = S3 = 1$

    $S2 = 0$

    C = initial value of semaphore *S*.

- *wait* operation

```
wait(S3);
wait(S1);
C := C - 1;
if C < 0
then begin
        signal(S1);
        signal(S2);
    end
else signal(S1);
signal(S3);
```

- *signal* operation

```
wait(S1);
C := C + 1;
if C ≤ 0 then signal(S2);
signal(S1);
```

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- Shared data

```
type item = ...
var  buffer = ...
     full, empty, mutex: semaphore;
     nextp, nextc: item;
     full := 0; empty := n ; mutex := 1;
```

- Producer process

```
repeat
      ...
    produce an item in nextp
      ...
   wait(empty);
   wait(mutex);
      ...
   add nextp to buffer
      ...
   signal(mutex);
   signal(full);
until false;
```

- Consumer process

```
repeat
    wait(full);
    wait(mutex);
        ...
    remove an item from buffer to nextc
        ...
    signal(mutex);
    signal(empty);
        ...
    consume the item in nextc
        ...
until false;
```

Readers–Writers Problem

- Shared data

```
var mutex, wrt: semaphore (= 1);
    readcount : integer (= 0);
```

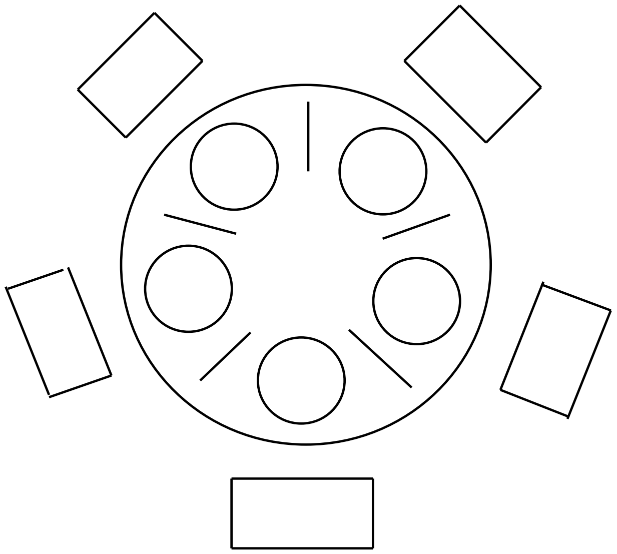
- Writer process

```
wait(wrt);
    ...
  writing is performed
    ...
signal(wrt);
```

- Reader process

```
wait(mutex);
   readcount := readcount + 1;
   if readcount = 1 then wait(wrt);
signal(mutex);
      ...
   reading is performed
      ...
wait(mutex);
   readcount := readcount – 1;
   if readcount = 0 then signal(wrt);
signal(mutex);
```

Dining-Philosophers Problem



- Shared data

  var *chopstick*: **array** [0..4] **of** *semaphore*;
  (=1 initially)

- Philosopher *i*:

```
repeat
    wait(chopstick[i]);
    wait(chopstick[i+1 mod 5]);
        ...
        eat
        ...
    signal(chopstick[i]);
    signal(chopstick[i+1 mod 5]);
        ...
        think
        ...
until false;
```

Critical Regions – high-level synchronization construct

- A shared variable *v* of type *T*, is declared as:

  var *v*: **shared** *T*

- Variable *v* accessed only inside statement:

  region *v* **when** *B* **do** *S*

  where *B* is a Boolean expression.

  While statement *S* is being executed, no other process can access variable *v*.

- Regions referring to the same shared variable exclude each other in time.

- When a process tries to execute the region statement, the Boolean expression *B* is evaluated. If *B* is true, statement *S* is executed. If it is false, the process is delayed until *B* becomes true and no other process is in the region associated with *v*.

Example – Bounded Buffer

- Shared variables:

```
var buffer: shared record
        pool: array [0..n–1] of item;
        count,in,out: integer;
    end;
```

- Producer process inserts *nextp* into the shared buffer

```
region buffer when count < n
    do begin
        pool[in] := nextp;
        in := in+1 mod n;
        count := count + 1;
    end;
```

- Consumer process removes an item from the shared buffer and puts it in *nextc*

```
region buffer when count > 0
    do begin
        nextc := pool[out];
        out := out+1 mod n;
        count := count – 1;
    end;
```

Implementation of:

region *x* **when** *B* **do** *S*

- We associate with the shared variable *x*, the following variables:

  ```
  var mutex, first-delay, second-delay: semaphore;
      first-count, second-count: integer;
  ```

- Mutually exclusive access to the critical section is provided by *mutex*.

- If a process cannot enter the critical section because the Boolean expression *B* is false, it initially waits on the *first-delay* semaphore; moved to the *second-delay* semaphore before it is allowed to reevaluate *B*.

- Keep track of the number of processes waiting on *first-delay* and *second-delay*, with *first-count* and *second-count* respectively.

- The Algorithm

```
wait(mutex);
while not B
    do begin
            first-count := first-count + 1;
            if second-count > 0
                   then signal(second-delay)
                   else signal(mutex);
            wait(first-delay);
            first-count := first-count - 1;
            second-count := second-count + 1;
            if first-count > 0
                  then signal(first-delay)
                  else signal(second-delay);
            wait(second-delay);
            second-count := second-count - 1;
          end;
S;
if first-count > 0
      then signal(first-delay);
      else if second-count > 0
                   then signal(second-delay);
                   else  signal(mutex);
```

- This algorithm assumes a FIFO ordering in the queueing of processes for a semaphore. For an arbitrary queueing discipline, a more complicated implementation is required.

Monitors – high-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
type monitor-name = monitor
    variable declarations

    procedure entry P1 ( ... );
        begin ... end;

    procedure entry P2 ( ... );
        begin ... end;
            .
            .
            .
    procedure entry Pn ( ... );
        begin ... end;

    begin
        initialization code
    end.
```

- To allow a process to wait within the monitor, a *condition* variable must be declared, as:

  var *x*,*y*: *condition*

- Condition variable can only be used with the operations *wait* and *signal*.

  - The operation

    x.wait;

    means that the process invoking this operation is suspended until another process invokes

    x.signal;

  - The *x.signal* operation resumes exactly one suspended process. If no process is suspended, then the *signal* operation has no effect.

```
type dining-philosophers = monitor
    var state : array [0..4] of (thinking, hungry, eating);
    var self : array [0..4] of condition;
    procedure entry pickup (i: 0..4);
        begin
            state[i] := hungry;
            test (i);
            if state[i] ≠ eating then self[i].wait;
        end;
    procedure entry putdown (i: 0..4);
        begin
            state[i] := thinking;
            test (i+4 mod 5);
            test (i+1 mod 5);
        end;
    procedure test (k: 0..4);
        begin
            if state[k+4 mod 5] ≠ eating
              and state[k] = hungry
              and state[k+1 mod 5] ≠ eating
              then begin
                        state[k] := eating;
                        self[k].signal;
                    end;
        end;
    begin
        for i := 0 to 4
            do state[i] := thinking;
    end.
```

Monitor implementation using semaphores.

- Variables

```
var mutex: semaphore (init = 1)
    next: semaphore (init = 0)
    next-count: integer (init = 0)
```

- Each external procedure *F* will be replaced by

```
wait(mutex);
    ...
  body of F;
    ...
if next-count > 0
  then signal(next)
  else signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

- For each condition variable *x*, we have:

```
var x-sem: semaphore (init = 0)
    x-count: integer (init = 0)
```

- The operation *x.wait* can be implemented as:

```
x-count := x-count + 1;
if next-count > 0
   then signal(next)
   else signal(mutex);
wait(x-sem);
x-count := x-count – 1;
```

- The operation *x.signal* can be implemented as:

```
if x-count > 0
   then begin
           next-count := next-count + 1;
           signal(x-sem);
           wait(next);
           next-count := next-count – 1;
        end;
```

- *Conditional-wait* construct

  $$x.wait(c);$$

  - *c* – integer expression evaluated when the wait operation is executed.
  - value of *c* (*priority number*) stored with the name of the process that is suspended.
  - when *x.signal* is executed, process with smallest associated priority number is resumed next.

- Must check two conditions to establish the correctness of this system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

Solaris 2 Operating System

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.
- Uses adaptive mutexes for efficiency when protecting data from short code segments.
- Uses condition variables and readers–writers locks when longer sections of code need access to data.

Atomic Transactions

- *Transaction* – program unit that must be executed atomically; that is, either all the operations associated with it are executed to completion, or none are performed.
- Must preserve atomicity despite possibility of failure.
- We are concerned here with ensuring transaction atomicity in an environment where failures result in the loss of information on volatile storage.

Log-Based Recovery

- *Write-ahead log* – all updates are recorded on the log, which is kept in stable storage; log has following fields:
  - transaction name
  - data item name, old value, new value

  The log has a record of <T_i **starts**>, and either <T_i **commits**> if the transactions commits, or <T_i **aborts**> if the transaction aborts.

- Recovery algorithm uses two procedures:
  - **undo**(T_i) – restores value of all data updated by transaction T_i to the old values. It is invoked if the log contains record <T_i **starts**>, but not <T_i **commits**>.
  - **redo**(T_i) – sets value of all data updated by transaction T_i to the new values. It is invoked if the log contains both <T_i **starts**> and <T_i **commits**>.

Checkpoints – reduce recovery overhead

1. Output all log records currently residing in volatile storage onto stable storage.
2. Output all modified data residing in volatile storage to stable storage.
3. Output log record <**checkpoint**> onto stable storage.

- Recovery routine examines log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place.
  - Search log backward for first <**checkpoint**> record.
  - Find subsequent <T_i **start**> record.
- **redo** and **undo** operations need to be applied to only transaction T_i and all transactions T_j that started executing after transaction T_i.

Concurrent Atomic Transactions

- *Serial schedule* – the transactions are executed sequentially in some order.

- Example of a serial schedule in which T_0 is followed by T_1:

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

- *Conflicting operations* – O_i and O_j *conflict* if they access the same data item, and at least one of these operations is a **write** operation.

- *Conflict serializable* schedule – schedule that can be transformed into a serial schedule by a series of swaps of nonconflicting operations.

- Example of a concurrent serializable schedule:

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

- *Locking protocol* governs how locks are acquired and released; data item can be locked in following modes:
  - **Shared:** If T_i has obtained a shared-mode lock on data item Q, then T_i can read this item, but it cannot write Q.
  - **Exclusive:** If T_i has obtained an exclusive-mode lock on data item Q, then T_i can both read and write Q.

- *Two-phase locking protocol*
  - **Growing phase:** A transaction may obtain locks, but may not release any lock.
  - **Shrinking phase:** A transaction may release locks, but may not obtain any new locks.

- The two-phase locking protocol ensures conflict serializability, but does not ensure freedom from deadlock.

- *Timestamp-ordering* scheme – transaction ordering protocol for determining serializability order.
  - With each transaction T_i in the system, associate a unique fixed timestamp, denoted by TS(T_i).
  - If T_i has been assigned timestamp TS(T_i), and a new transaction T_j enters the system, then TS(T_i) < TS(T_j).
- Implement by assigning two timestamp values to each data item Q.
  - **W-timestamp**(Q) – denotes largest timestamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) – denotes largest timestamp of any transaction that executed **read**(Q) successfully.

- Example of a schedule possible under the timestamp protocol:

T_2	T_3
read(B)	
	read(B)
	write(B)
read(A)	
	read(A)
	write(A)

- There are schedules that are possible under the two-phase locking protocol but are not possible under the timestamp protocol, and vice versa.

- The timestamp-ordering protocol ensures conflict serializability; conflicting operations are processed in timestamp order.

CHAPTER 7: DEADLOCKS

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

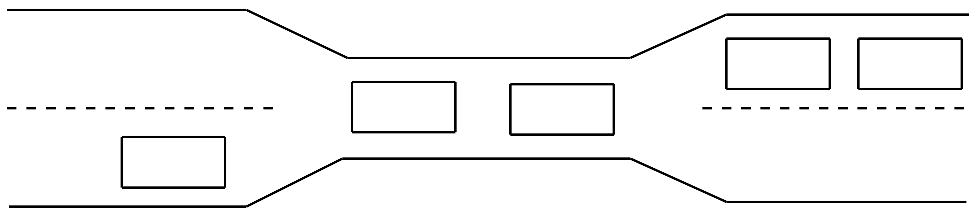
- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.

- Example

  semaphores A and B, initialized to 1

P_0	P_1
$wait(A)$	$wait(B)$
$wait(B)$	$wait(A)$

- Example: bridge crossing



  - Traffic only in one direction.

  - Each section of a bridge can be viewed as a resource.

  - If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).

  - Several cars may have to be backed up if a deadlock occurs.

  - Starvation is possible.

System Model

- Resource types $R_1, R_2, ..., R_{m-1}$

  CPU cycles, memory space, I/O devices

- Each resource type R_i has W_i instances.

- Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterization – deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource.

- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.

- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.

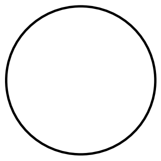
- **Circular wait:** there exists a set $\{P_0, P_1, ..., P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Resource-Allocation Graph – a set of vertices V and a set of edges E.

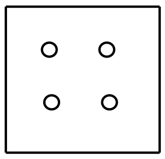
- V is partitioned into two types:
  - $P = \{P_1, P_2, ..., P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, ..., R_m\}$, the set consisting of all resource types in the system.
- *request edge* – directed edge $P_i \rightarrow R_j$
- *assignment edge* – directed edge $R_j \rightarrow P_i$

Example

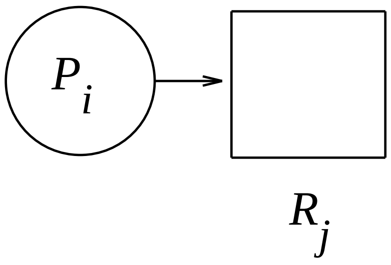
- Process



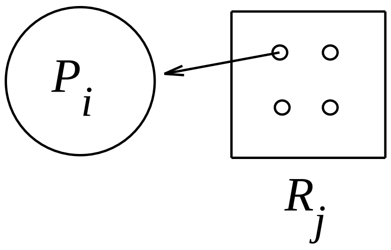
- Resource type with 4 instances



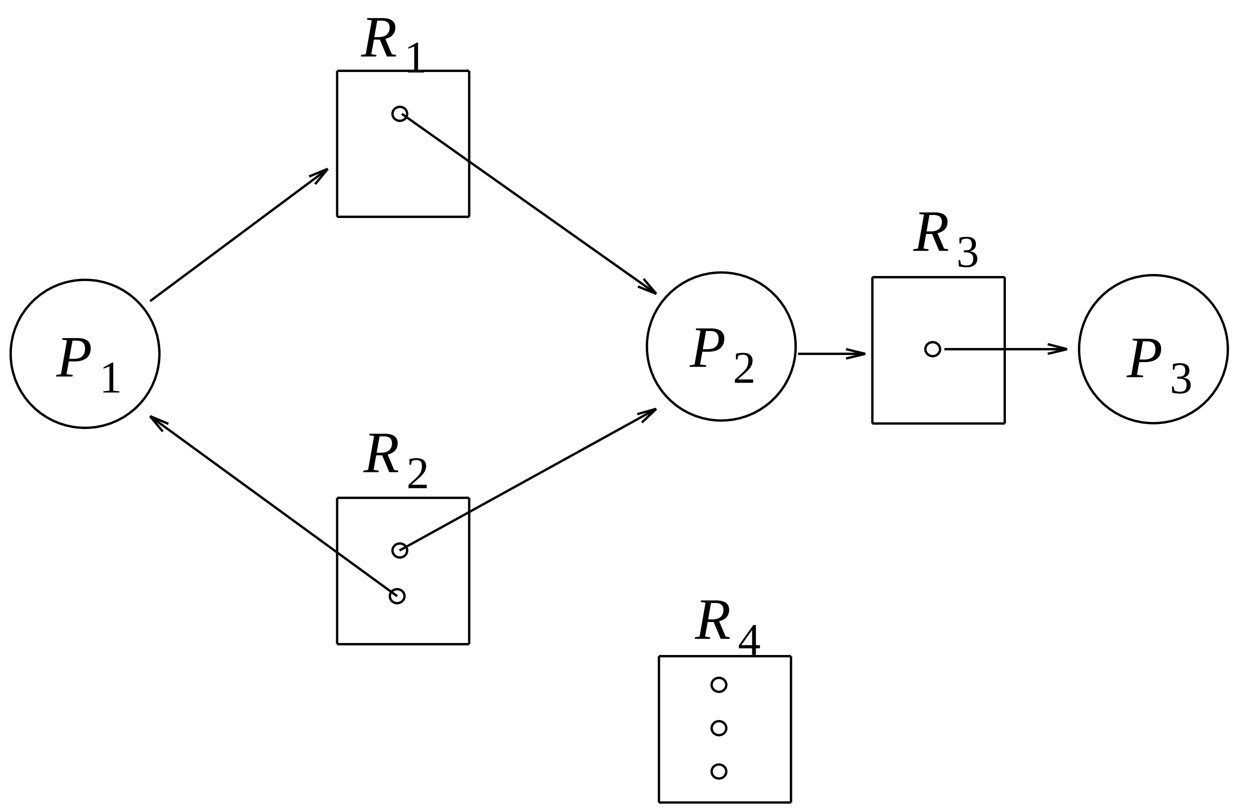
- P_i requests instance of R_j



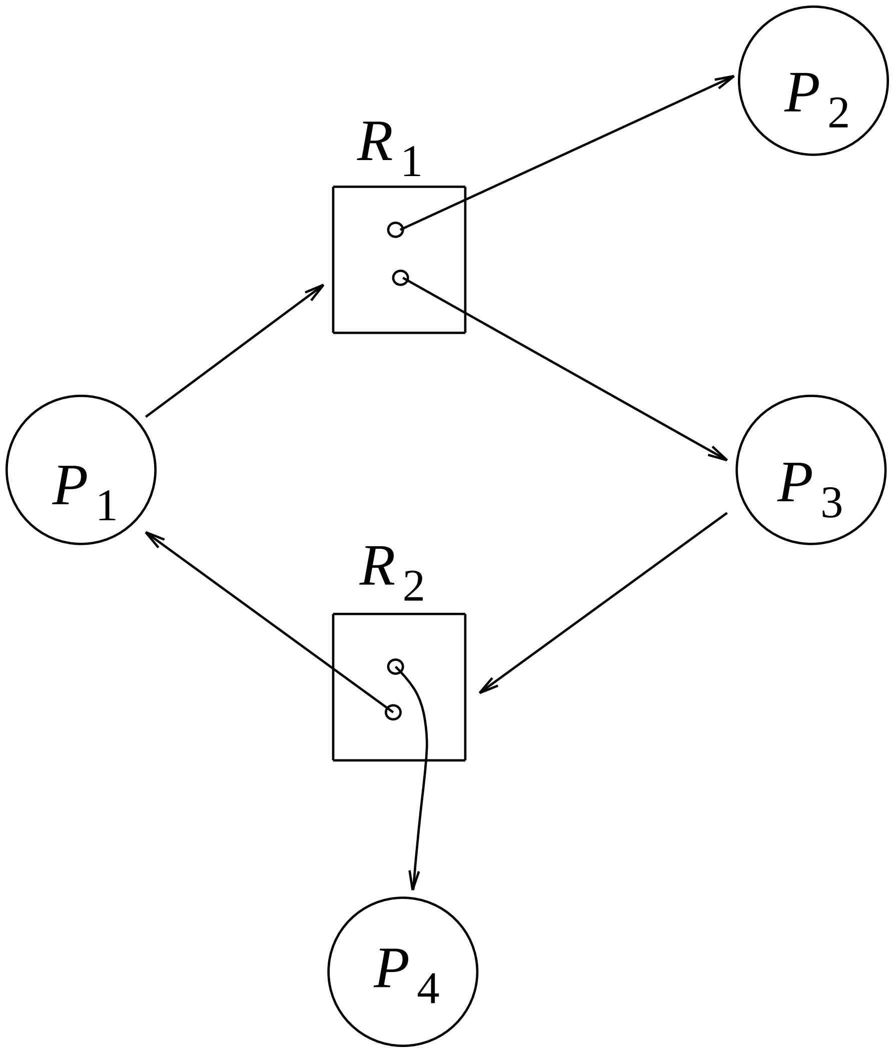
- P_i is holding an instance of R_j



- Example of a resource-allocation graph with no cycles.



- Example of a resource-allocation graph with a cycle.



- If graph contains no cycles ⇒ no deadlock.

- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention – restrain the ways resource requests can be made.

- Mutual Exclusion – not required for sharable resources; must hold for nonsharable resources.

- Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible.

- No Preemption –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Preempted resources are added to the list of resources for which the process is waiting.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

- Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance – requires that the system has some additional *a priori* information available.

- Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.

- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State – when a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

- System is in safe state if there exists a *safe sequence* of all processes.

- Sequence $<P_1, P_2, ..., P_n>$ is safe if for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $j < i$.

  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished.

  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.

  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

- If a system is in safe state $\Rightarrow$ no deadlocks.

- If a system is in unsafe state $\Rightarrow$ possibility of deadlock.

- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j; represented by a dashed line.

- Claim edge converts to request edge when a process requests a resource.

- When a resource is released by a process, assignment edge reconverts to a claim edge.

- Resources must be claimed *a priori* in the system.

Banker’s Algorithm

- Multiple resource types.
- Each process must *a priori* claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

- Data Structures for the Banker's algorithm where n = number of processes, and m = number of resource types.

  - *Available:* Vector of length m. If *Available*[j] = k, there are k instances of resource type R_j available.

  - *Max:* $n \times m$ matrix. If *Max*[i,j] = k, then process P_i may request at most k instances of resource type R_j.

  - *Allocation:* $n \times m$ matrix. If *Allocation*[i,j] = k, then P_i is currently allocated k instances of R_j.

  - *Need:* $n \times m$ matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task.

$$Need[i,j] = Max[i,j] - Allocation[i,j].$$

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively.

Initialize:

```
    Work := Available
    Finish[i] := false for i = 1, 2, ..., n.
```

2. Find an i such that both:

   a. *Finish*[*i*] = *false*

   b. $Need_i \leq Work$

   If no such i exists, go to step 4.

3. *Work* := *Work* + $Allocation_i$

   Finish[*i*] := *true*

   go to step 2.

4. If *Finish*[*i*] = *true* for all *i*, then the system is in a safe state.

May require an order of $m \times n^2$ operations to decide whether a state is safe.

Resource-Request Algorithm for process P_i

$Request_i$ = request vector for process P_i.

If $Request_i[j] = k$, then process P_i wants k instances of resource type R_j.

1. If $Request_i \leq Need_i$, go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.

2. If $Request_i \leq Available$, go to step 3. Otherwise, P_i must wait, since resources are not available.

3. Pretend to allocate requested resources to P_i by modifying the state as follows:

   $Available := Available - Request_i;$
   $Allocation_i := Allocation_i + Request_i;$
   $Need_i := Need_i - Request_i;$

   - If safe $\Rightarrow$ the resources are allocated to P_i.
   - If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored.

Example of Banker’s algorithm

- 5 processes P_0 through P_4; 3 resource types A (10 instances), B (5 instances), and C (7 instances).

- Snapshot at time T_0:

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

- Sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies safety criteria.

- P_1 now requests resources.

  $Request_1 = (1,0,2)$.

  - Check that $Request_1 \leq Available$ (that is, $(1,0,2) \leq (3,3,2)) \Rightarrow$ true.

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

  - Executing safety algorithm shows that sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies safety requirement.

- Can request for (3,3,0) by P_4 be granted?

- Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes.
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Several Instances of a Resource Type

- Data structures

  - *Available:* A vector of length m indicates the number of available resources of each type.

  - *Allocation:* An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.

  - *Request:* An $n \times m$ matrix indicates the current request of each process. If *Request*$[i,j] = k$, then process P_i is requesting k more instances of resource type R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize:
   - $Work := Available$.
   - For $i = 1, 2, ..., n$, if $Allocation_i \neq$, then $Finish[i] := false$; otherwise, $Finish[i] := true$.
2. Find an index i such that both:
   a. $Finish[i] = false$.
   b. $Request_i \leq Work$.
   If no such i exists, go to step 4.
3. $Work := Work + Allocation_i$
   $Finish[i] := true$
   go to step 2.
4. If $Finish[i]$ = false, for some i, $1 \leq i \leq n$, then the system is in a deadlock state. Moreover, if $Finish[i] = false$, then P_i is deadlocked.

- Algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

Example of Detection algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).

- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence $<P_0, P_2, P_3, P_1, P_4>$ will result in *Finish*[*i*] = true for all *i*.

- P_2 requests an additional instance of type C.

	Request
	$A\ B\ C$
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes' requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle

- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes ‘‘caused’’ the deadlock.

Recovery from Deadlock

- Process termination
  - Abort all deadlocked processes.
  - Abort one process at a time until the deadlock cycle is eliminated.
  - In which order should we choose to abort?
    - Priority of the process.
    - How long process has computed, and how much longer to completion.
    - Resources the process has used.
    - Resources process needs to complete.
    - How many processes will need to be terminated.
    - Is process interactive or batch?

- Resource Preemption
  - Selecting a victim – minimize cost.
  - Rollback – return to some safe state, restart process from that state.
  - Starvation – same process may always be picked as victim; include number of rollback in cost factor.

Combined Approach to Deadlock Handling

- Combine the three basic approaches (prevention, avoidance, and detection), allowing the use of the optimal approach for each class of resources in the system.

- Partition resources into hierarchically ordered classes.

- Use most appropriate technique for handling deadlocks within each class.

CHAPTER 8: MEMORY MANAGEMENT

- Background
- Logical versus Physical Address Space
- Swapping
- Contiguous Allocation
- Paging
- Segmentation
- Segmentation with Paging

Background

- Program must be brought into memory and placed within a process for it to be executed.
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory for execution.
- User programs go through several steps before being executed.

Address binding of instructions and data to memory addresses can happen at three stages:

- **Compile time:** If memory location known a priori, *absolute* code can be generated; must recompile code if starting location changes.

- **Load time:** Must generate *relocatable* code if memory location is not known at compile time.

- **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit* registers).

- Dynamic Loading – routine is not loaded until it is called.
  - Better memory-space utilization; unused routine is never loaded.
  - Useful when large amounts of code are needed to handle infrequently occurring cases.
  - No special support from the operating system is required; implemented through program design.

- Dynamic Linking – linking postponed until execution time.
  - Small piece of code, *stub*, used to locate the appropriate memory-resident library routine.
  - Stub replaces itself with the address of the routine, and executes the routine.
  - Operating system needed to check if routine is in processes' memory address.

- Overlays – keep in memory only those instructions and data that are needed at any given time.
  - Needed when process is larger than amount of memory allocated to it.
  - Implemented by user, no special support needed from operating system; programming design of overlay structure is complex.

Logical versus Physical Address Space

- The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management.
  - *Logical address* – generated by the CPU; also referred to as *virtual address*.
  - *Physical address* – address seen by the memory unit.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

Memory-management unit (MMU) – hardware device that maps virtual to physical address.

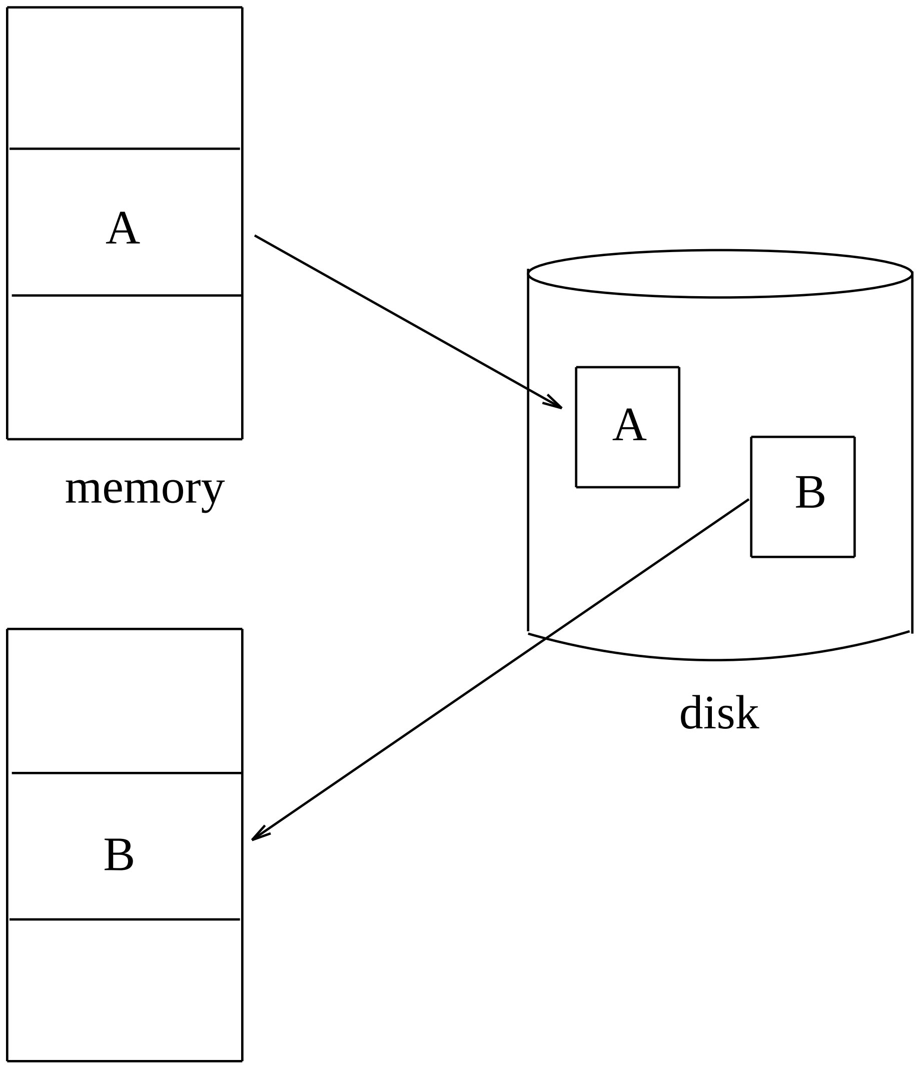
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory.

- The user program deals with *logical* addresses; it never sees the *real* physical addresses.

Swapping

- A process can be *swapped* temporarily out of memory to a *backing store,* and then brought back into memory for continued execution.
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
- *Roll out, roll in* – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.
- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.
- Modified versions of swapping are found on many systems, i.e., UNIX and Microsoft Windows.

- Schematic view of swapping

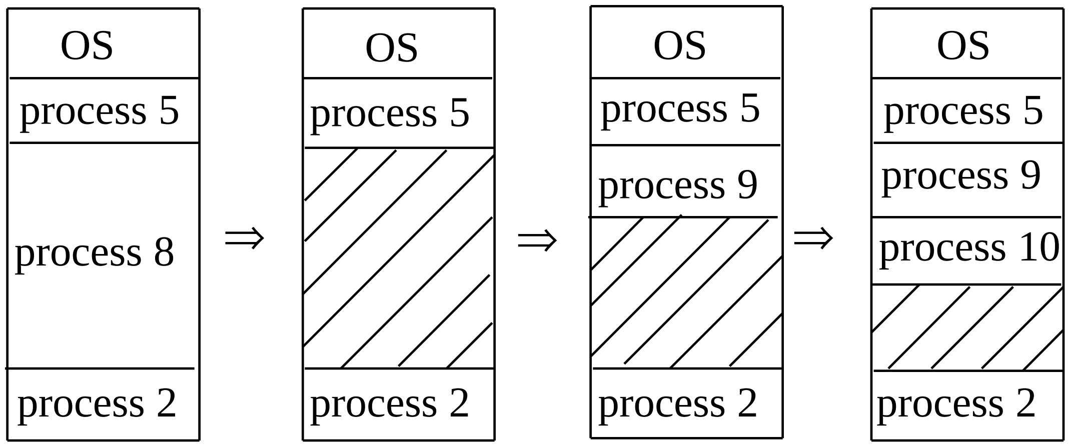


Contiguous Allocation

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector.
  - User processes then held in high memory.
- Single-partition allocation
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.

- Multiple-partition allocation
  - *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.

- Example



Operating system maintains information about:

- allocated partitions
- free partitions (hole)

- *Dynamic storage-allocation* problem – how to satisfy a request of size *n* from a list of free holes.
  - **First-fit:** Allocate the *first* hole that is big enough.
  - **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
  - **Worst-fit:** Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

- First-fit and best-fit better than worst-fit in terms of speed and storage utilization.

- External fragmentation – total memory space exists to satisfy a request, but it is not contiguous.

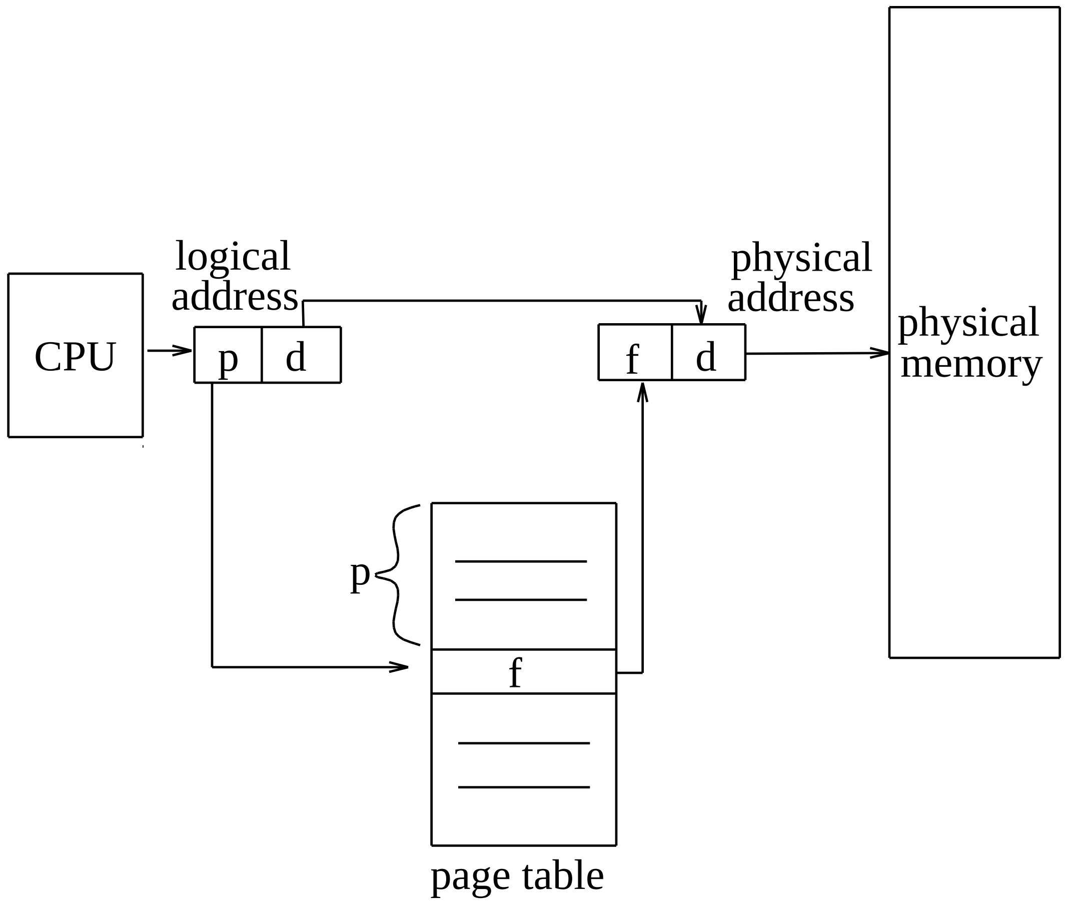
- Internal fragmentation – allocated memory may be slightly larger than requested memory; difference between these two numbers is memory internal to a partition, but not being used.

- Reduce external fragmentation by *compaction.*
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time.
  - I/O problem
    - Latch job in memory while it is involved in I/O.
    - Do I/O only into OS buffers.

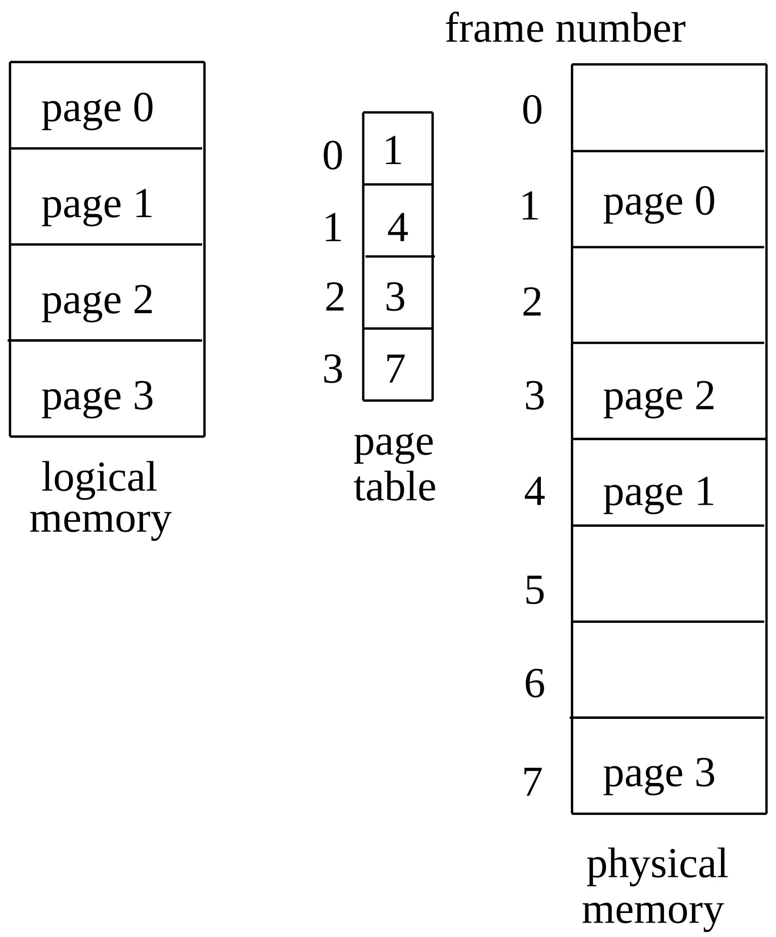
Paging – logical address space of a process can be noncontiguous; process is allocated physical memory wherever the latter is available.

- Divide physical memory into fixed-sized blocks called *frames* (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size called *pages*.
- Keep track of all free frames.
- To run a program of size *n* pages, need to find *n* free frames and load program.
- Set up a page table to translate logical to physical addresses.
- Internal fragmentation.

- Address generated by CPU is divided into:
  - *Page number (p)* – used as an index into a *page table* which contains base address of each page in physical memory.
  - *Page offset (d)* – combined with base address to define the physical memory address that is sent to the memory unit.



- Separation between user's view of memory and actual physical memory reconciled by address-translation hardware; logical addresses are translated into physical addresses.



Implementation of page table

- Page table is kept in main memory.
- *Page-table base register (PTBR)* points to the page table.
- *Page-table length register (PTLR)* indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative registers* or *translation look-aside buffers (TLBs).*

- Associative registers – parallel search

Page #	Frame #

Address translation (A′, A″)

- If A′ in associative register, get frame # out.
- Otherwise get frame # from page table in memory.

- *Hit ratio* – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers.

- Effective Access Time (EAT)
  - associative lookup = ε time unit
  - memory cycle time - 1 microsecond
  - hit ratio = α

$$\begin{aligned} \text{EAT} &= (1 + \varepsilon)\,\alpha + (2 + \varepsilon)\,(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$

- Memory protection implemented by associating protection bits with each frame.

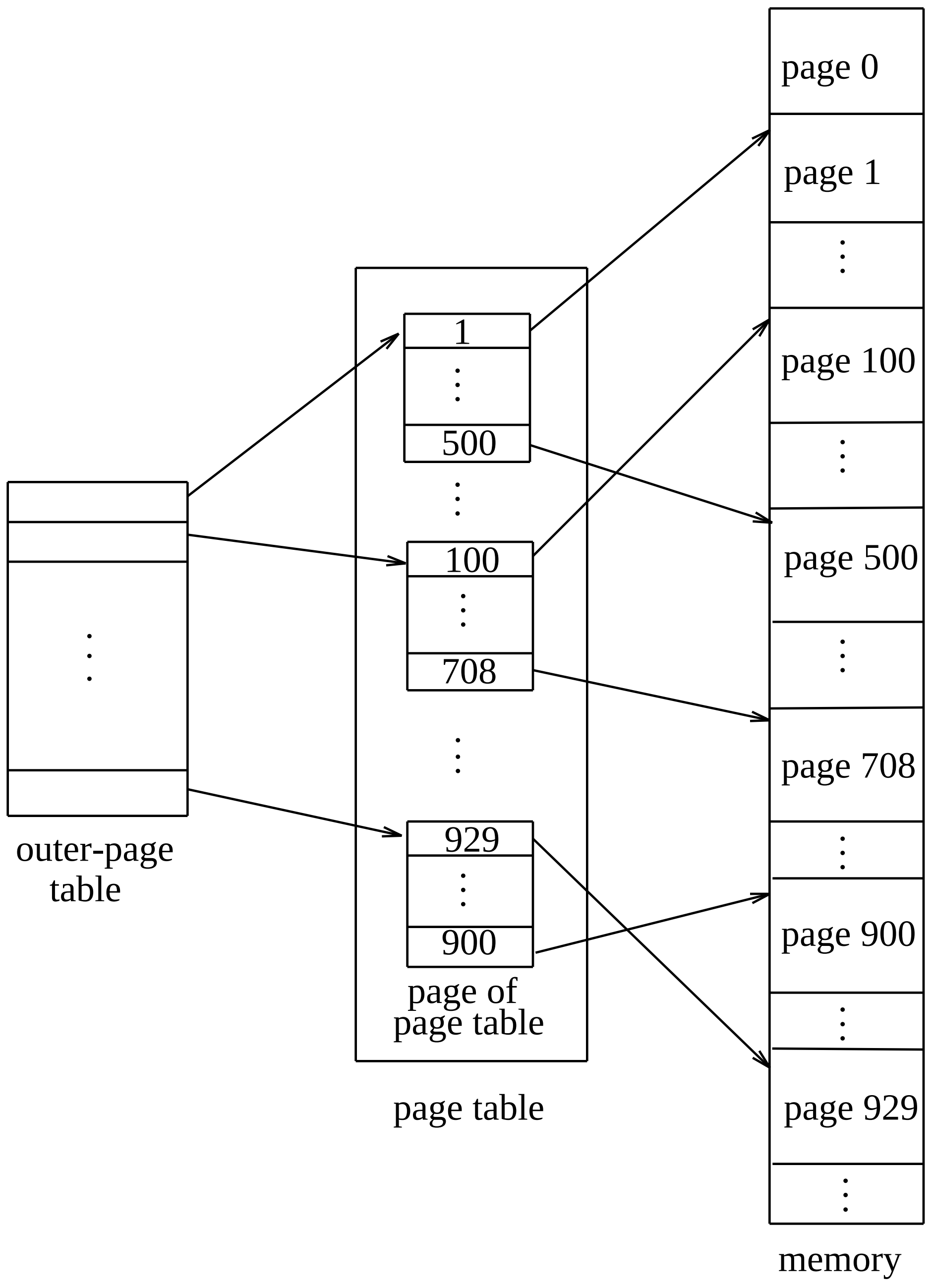
- *Valid–invalid* bit attached to each entry in the page table:

  - ‘‘valid’’ indicates that the associated page is in the process’ logical address space, and is thus a legal page.

  - ‘‘invalid’’ indicates that the page is not in the process’ logical address space.

Multilevel Paging – partitioning the page table allows the operating system to leave partitions unused until a process needs them.

- A two-level page-table scheme



- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.

- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.

- Thus, a logical address is as follows:

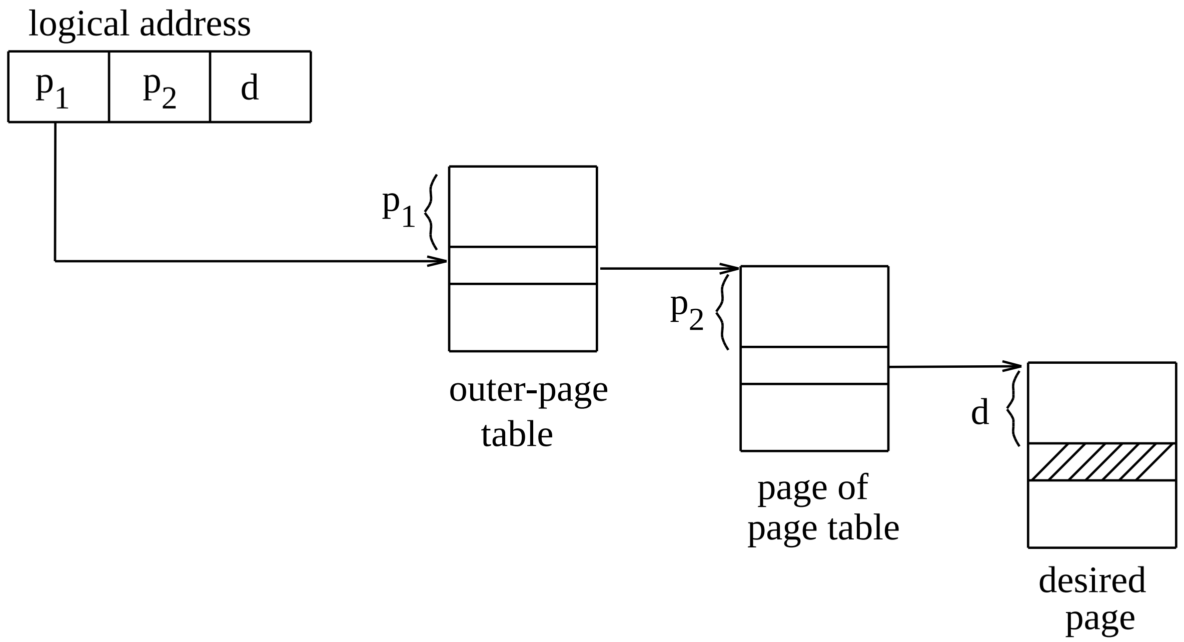
page number		page offset
p_1	p_2	d
10	10	12

where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table.

- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

- Address-translation scheme for a two-level 32-bit paging architecture



- Multilevel paging and performance

  - Since each level is stored as a separate table in memory, converting a logical address to a physical one may take four memory accesses.

  - Even though time needed for one memory access is quintupled, caching permits performance to remain reasonable.

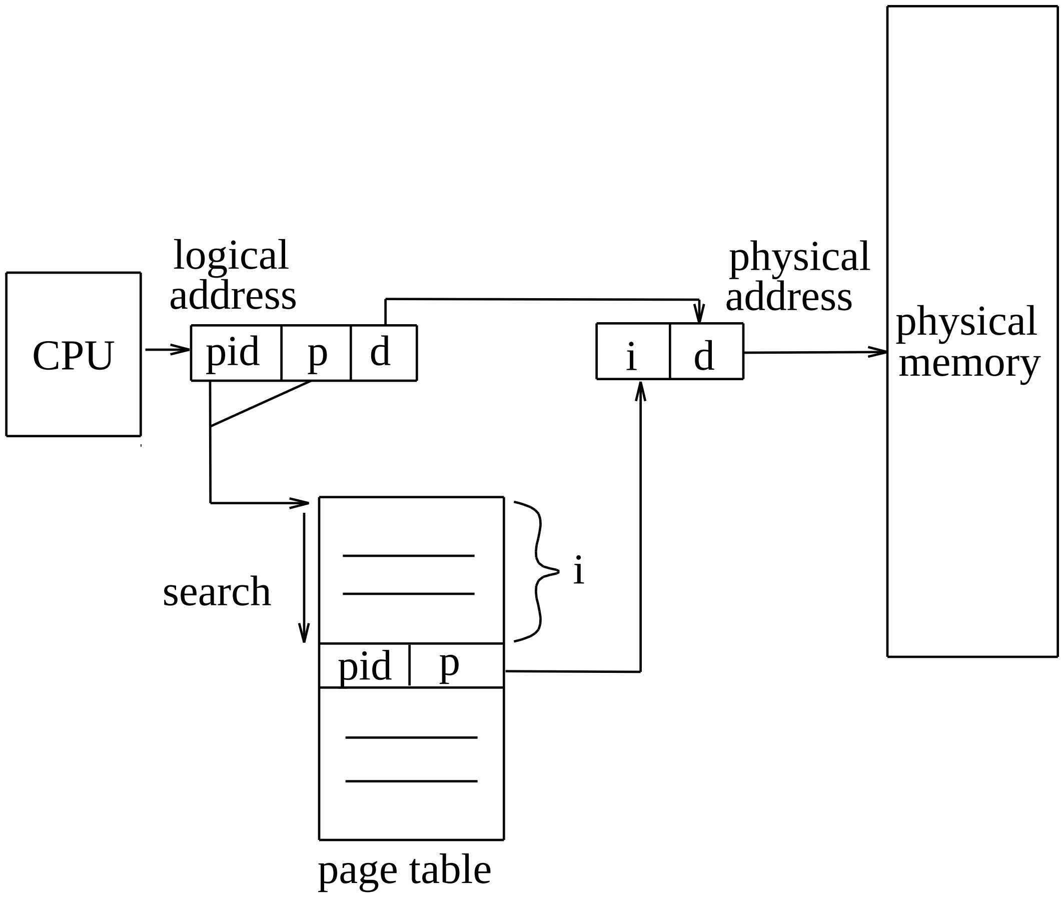
  - Cache hit rate of 98 percent yields:

$$\begin{aligned}\text{effective access time} &= 0.98 \times 120 + 0.02 \times 520 \\ &= 128 \text{ nanoseconds}\end{aligned}$$

    which is only a 28 percent slowdown in memory access time.

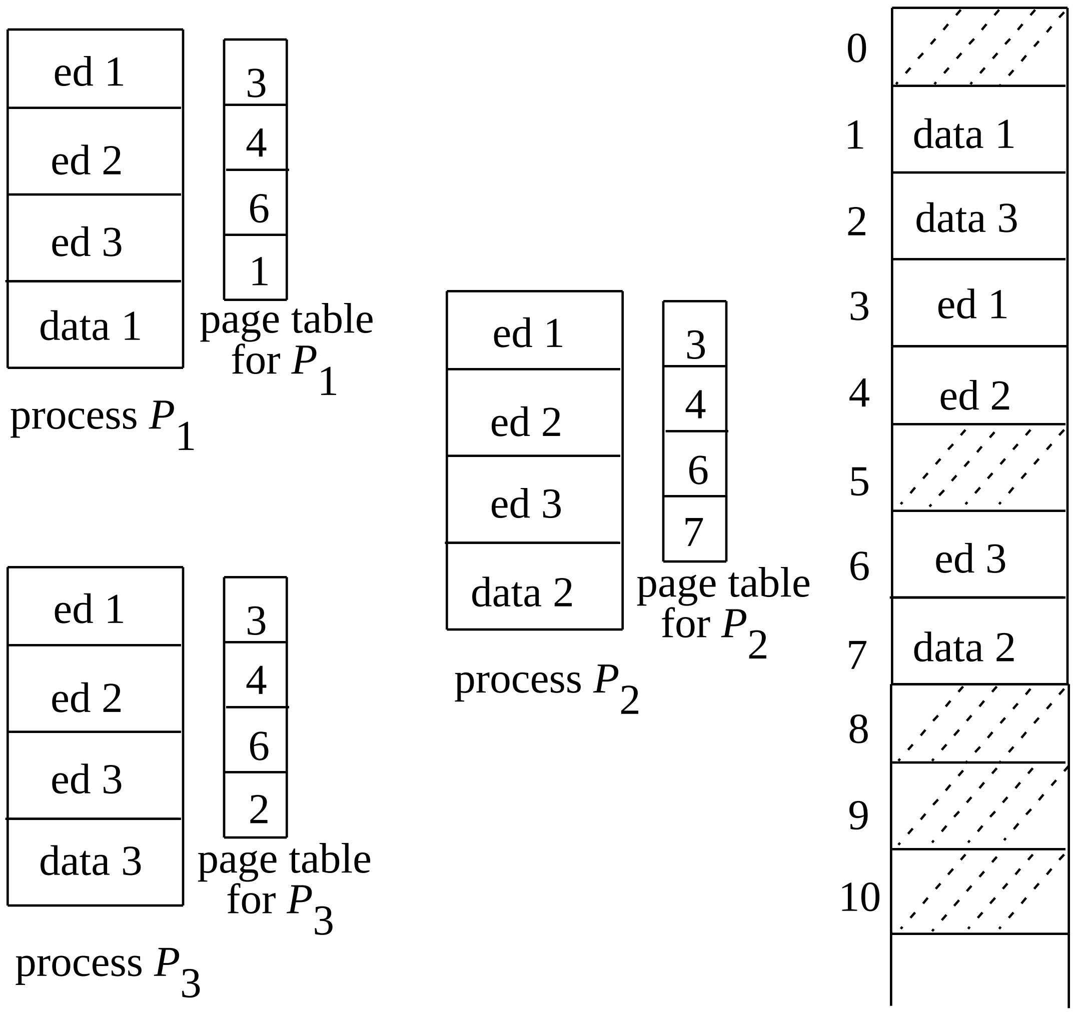
Inverted Page Table – one entry for each real page of memory; entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

- Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.



Shared pages

- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).

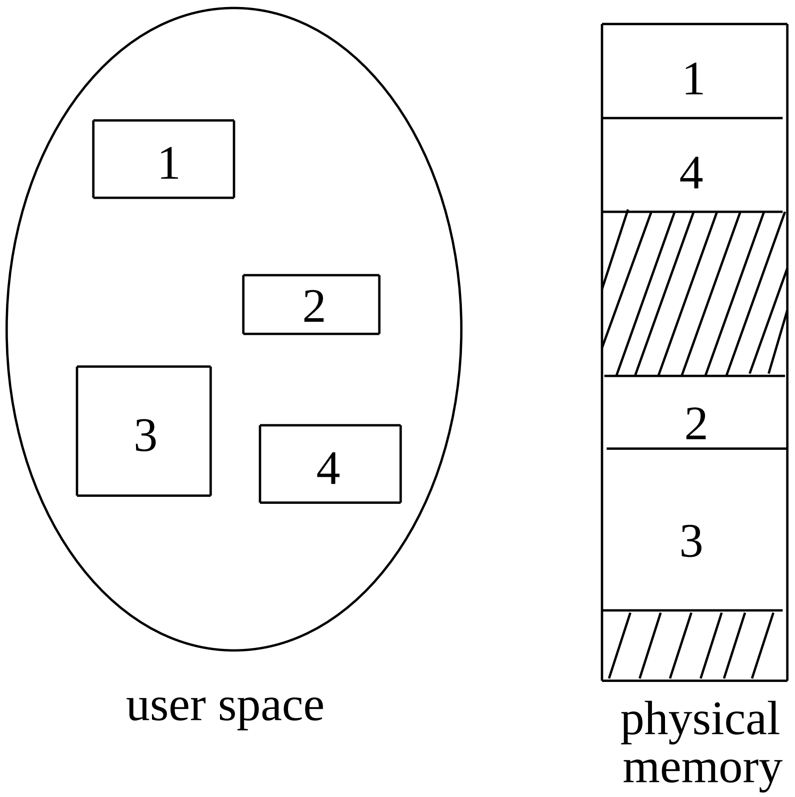


Segmentation – memory-management scheme that supports user view of memory.

- A program is a collection of segments. A segment is a logical unit such as:

  main program
  procedure
  function
  local variables, global variables
  common block
  stack
  symbol table, arrays

- Example



- Logical address consists of a two tuple:

  <segment-number, offset>.

- *Segment table* – maps two-dimensional user-defined addresses into one-dimensional physical addresses; each entry of table has:
  - *base* – contains the starting physical address where the segments reside in memory.
  - *limit* – specifies the length of the segment.

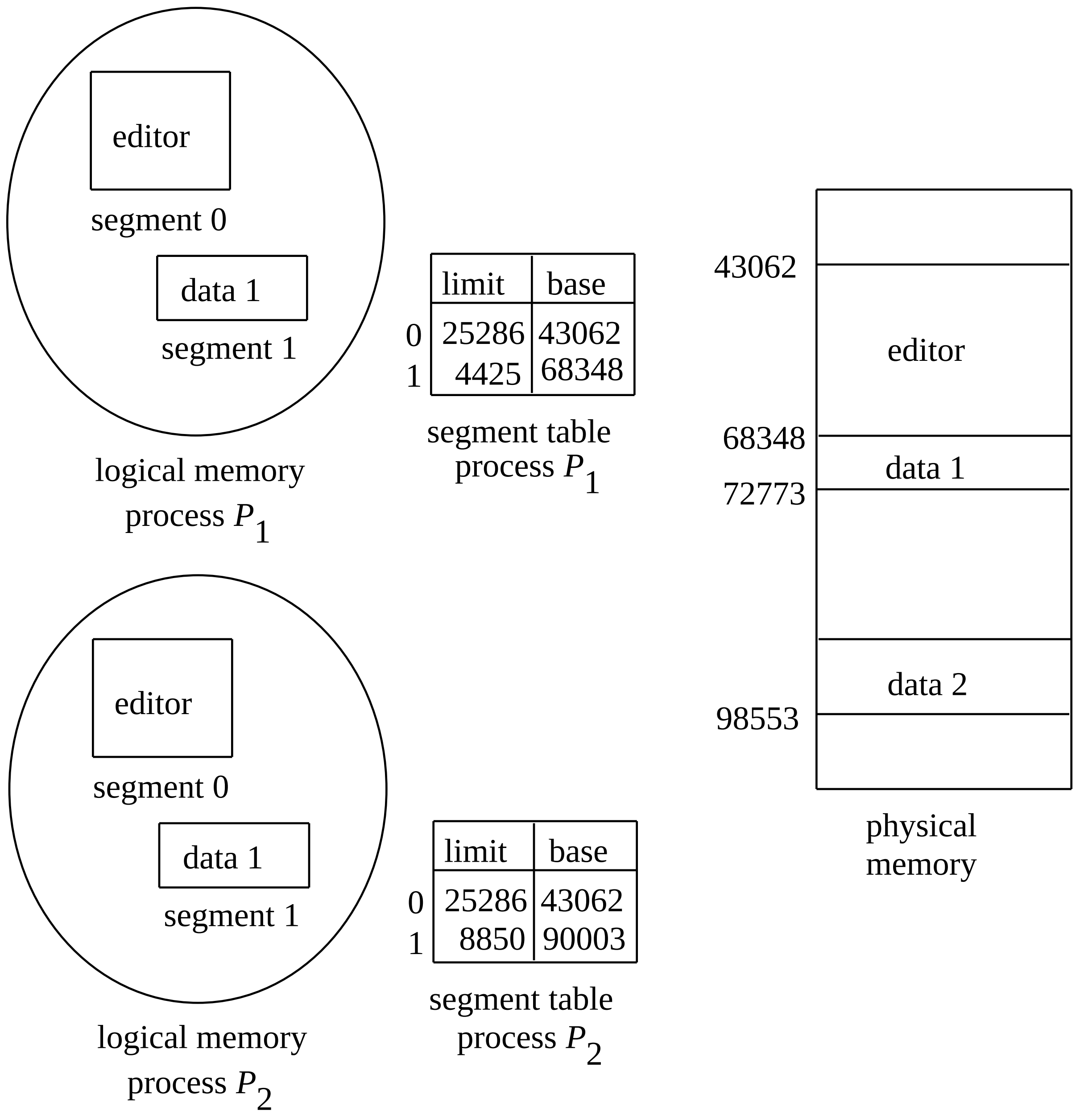
- *Segment-table base register (STBR)* points to the segment table's location in memory.

- *Segment-table length register (STLR)* indicates number of segments used by a program;

  segment number s is legal if $s <$ STLR.

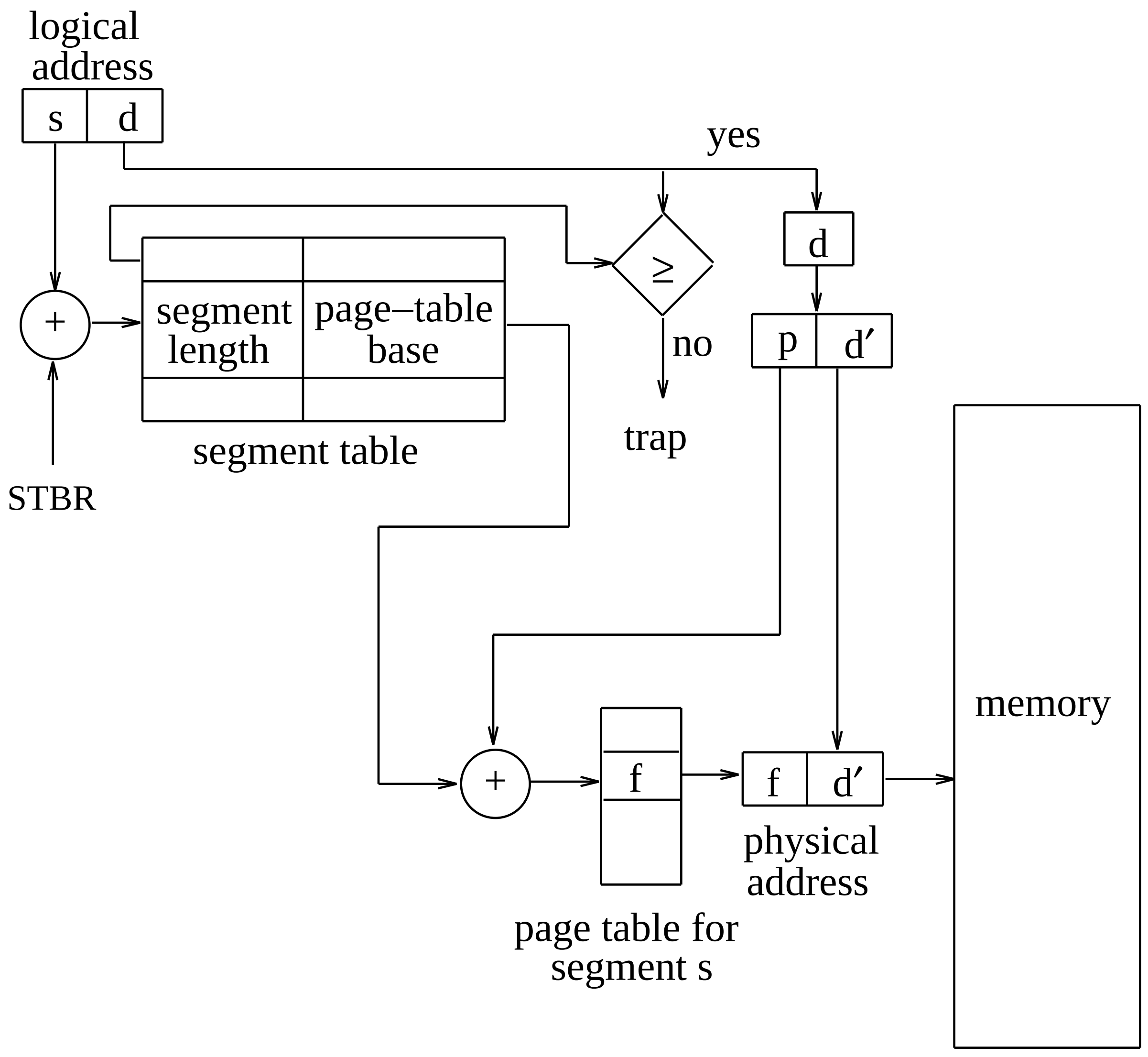
Relocation	- dynamic
	- by segment table
Sharing	- shared segments
	- same segment number
Protection	With each entry in segment table associate:
	- validation bit = 0 $\Rightarrow$ illegal segment
	- read/write/execute privileges
Allocation	- first fit/best fit
	- external fragmentation

- Protection bits associated with segments; code sharing occurs at segment level.
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem.



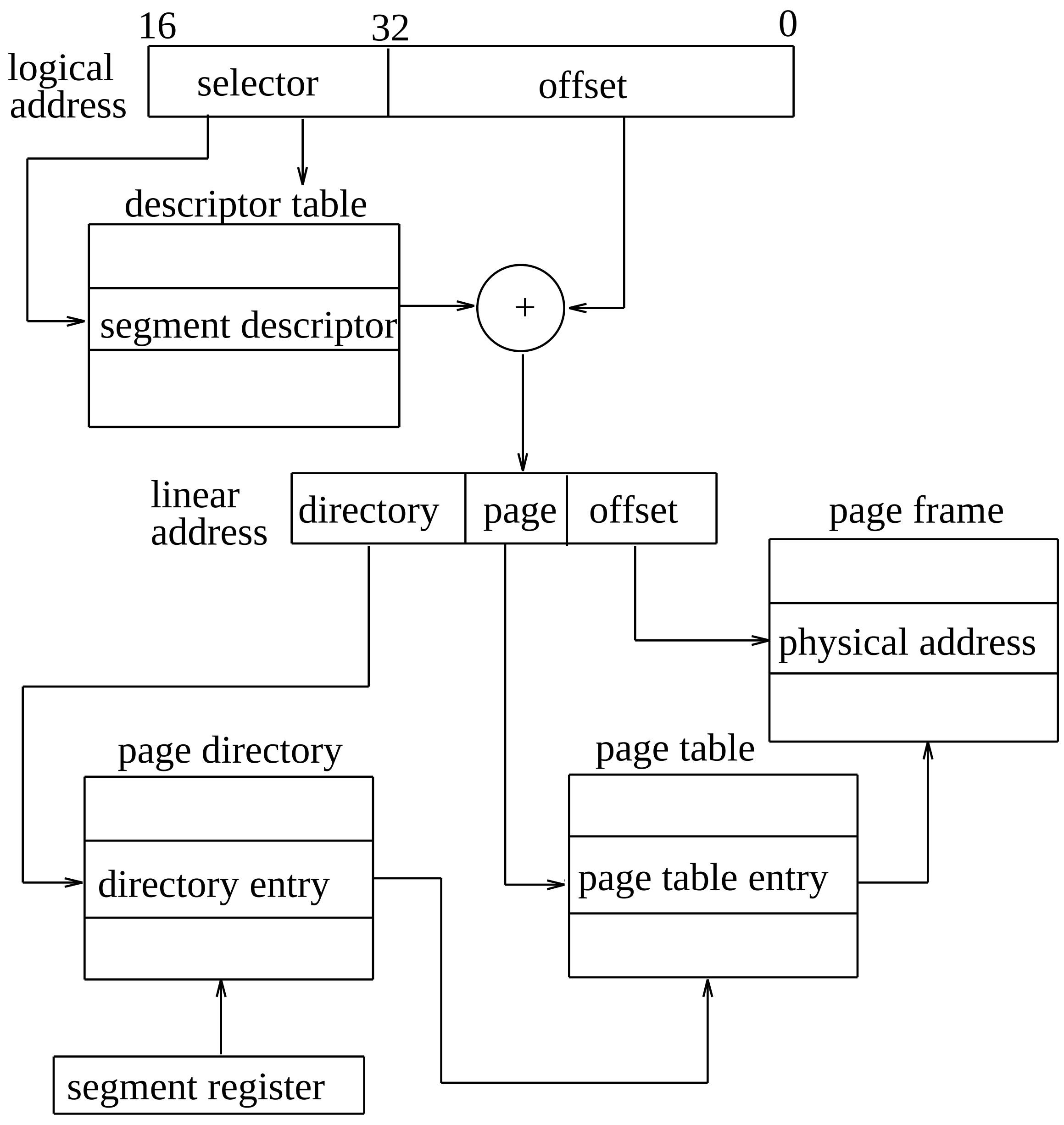
Segmentation with Paging

- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.



Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment.

- The Intel 386 uses segmentation with paging for memory management, with a two-level paging scheme.



Considerations in comparing memory-management strategies:

- Hardware support
- Performance
- Fragmentation
- Relocation
- Swapping
- Sharing
- Protection