CS 332/532 Systems Programming

Lecture 34

Semaphore &

Memory Management

Professor : Mahmut Unan – UAB CS

sem_init

- **#include <semaphore.h>**
- **int sem_init(sem_t ****sem***, int** *pshared***, unsigned int** *value***);**
- Link with *-pthread*.

- **sem_init**() initializes the unnamed semaphore at the address pointed to by *sem*. The *value* argument specifies the initial value for the semaphore. The *pshared* argument indicates whether this semaphore is to be shared between the threads of a process, or between processes. If *pshared* has the value 0, then the semaphore is shared between the threads of a process, and should be located at some address that is visible to all threads (e.g., a global variable, or a variable allocated dynamically on the heap).

https://www.man7.org/linux/man-pages/man3/sem_init.3.html

Thread 1	Thread 2	data
sem_wait (&mutex);	---	0
---	sem_wait (&mutex);	0
a = data;	/* blocked */	0
a = a+1;	/* blocked */	0
data = a;	/* blocked */	1
sem_post (&mutex);	/* blocked */	1
/* blocked */	b = data;	1
/* blocked */	b = b - 1;	1
/* blocked */	data = b;	2
/* blocked */	sem_post (&mutex);	2

http://www.csc.villanova.edu/~mdamian/threads/posixsem.html

- We will implement the bounded-buffer producer-consumer problem using semaphores here. In this exercise we will consider the case of a single producer and single consumer and use threads to create a producer and a consumer.
- We use the pseudo code from the textbook (Figure 5.16 on page 228) and replace semWait and semSignal with *sem_wait* and *sem_post*.
- This code is based on the examples provided in the classic book - UNIX Networking Programming, Volume 2 by W. Richard Stevens

prodcons1.c

```c
/* Solution to the single Producer/Consumer problem using semaphores.
   This example uses a circular buffer to put and get the data
   (a bounded-buffer).
   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens

   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
   To run: ./<filename> <#items>

   To enable printing add -DDEBUG to compile:
   gcc -O -Wall -DDEBUG -o <filename> <filename>.c -lpthread
*/

/* include globals */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>     /* for POSIX threads */
#include <semaphore.h>   /* for POSIX semaphores */

```

```c

#define  NBUFF           10

int      nitems;                                /* read-only */

struct {                    /* data shared by producer and consumer */
  int       buff[NBUFF];
  sem_t     mutex, nempty, nstored;   /* semaphores, not pointers */
} shared;

void       *producer(void *), *consumer(void *);

/* end globals */

/* main program */
int main(int argc, char **argv)
{
  pthread_t       tid_producer, tid_consumer;

  if (argc != 2) {
    printf("Usage: %s <#items>\n", argv[0]);
    exit(-1);
  }

```

```c

  nitems = atoi(argv[1]);

  /* initialize three semaphores */
  sem_init(&shared.mutex, 0, 1);
  sem_init(&shared.nempty, 0, NBUFF);
  sem_init(&shared.nstored, 0, 0);

  /* create one producer thread and one consumer thread */
  pthread_create(&tid_producer, NULL, producer, NULL);
  pthread_create(&tid_consumer, NULL, consumer, NULL);

  /* wait for producer and consumer threads */
  pthread_join(tid_producer, NULL);
  pthread_join(tid_consumer, NULL);

  /* remove the semaphores */
  sem_destroy(&shared.mutex);
  sem_destroy(&shared.nempty);
  sem_destroy(&shared.nstored);

  return 0;
}
/* end main */
```

```c
/* producer function */
void *producer(void *arg)
{
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
    sem_wait(&shared.mutex);

    shared.buff[i % NBUFF] = i;     /* store i into circular buffer */
#ifdef DEBUG
    printf("wrote %d to buffer at location %d\n", i, i % NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nstored);      /* 1 more stored item */
  }
  return (NULL);
}
/* end producer */
```

```c
/* consumer function */
void *consumer(void *arg)
{
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nstored);       /* wait for at least 1 stored item */
    sem_wait(&shared.mutex);

    if (shared.buff[i % NBUFF] != i)
      printf("error: buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
#ifdef DEBUG
    printf("read %d from buffer at location %d\n",
            shared.buff[i % NBUFF], i % NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nempty);        /* 1 more empty slot */
  }
  return (NULL);
}
/* end consumer */
```

- The source code is self-explanatory, we will focus on key sections of the code here.
- First, we define global variables such as the number of items (*nitems*) the producer will produce and the consumer will consume.
- Then we create a shared region, called *shared*, that will be shared between the producer and consumer threads.
- It contains the buffer shared by the producer and consumer and the three semaphores: one for the mutex lock (*mutex*), one for number of empty slots (*nempty*), and one for the number of slots filled (*nstored*) (these correspond to semaphores *s*, *e*, and *n*, respectively, from the textbook).

```
int       nitems;          /* read-only */
struct {
/* data shared by producer and consumer */
   int        buff[NBUFF];
    sem_t     mutex, nempty, nstored;
      /* semaphores */
} shared;
```

- In the main function, we read the number of items to be produced/consumed as a command-line argument and initialize the three semaphores using *sem_init* as per the pseudocode from the textbook.

```
nitems = atoi(argv[1]);

/* initialize three semaphores */
sem_init(&shared.mutex, 0, 1);
sem_init(&shared.nempty, 0, NBUFF);
sem_init(&shared.nstored, 0, 0);
```

- We create two separate threads, one for the producer and one for the consumer, and wait for the two threads to complete.

```
/* create one producer thread and one
consumer thread */
pthread_create(&tid_producer, NULL,
producer, NULL);
pthread_create(&tid_consumer, NULL,
consumer, NULL);

/* wait for producer and consumer threads */
pthread_join(tid_producer, NULL);
pthread_join(tid_consumer, NULL);
```

- Now let us look at the producer thread.
- It executes a loop equal to the number of items specified (*nitems*) and during each iteration of the loop, waits on the semaphore *nempty*.
- Initially *nempty* is set to *NBUFF*, so *sem_wait* returns immediately and waits on the semaphore *mutex*.
  Since *mutex* is initially set to 1, the producer thread enters the critical section and assigns the value *i* to the buffer location *i % NBUFF* and then release the *mutex* semaphore (calls *sem_post* on the semaphore *mutex*).
- Now that there is at least one element in the buffer, it also posts *sem_post* on the semaphore *nstored* to indicate to the consumer that there is an element in the buffer and continues with the loop.
- The producer thread terminates when the loop completes (i.e., after *nitems* iterations).
-

- ```
  /* producer function */
  void *producer(void *arg) {
    int i;

    for (i = 0; i < nitems; i++) {
      sem_wait(&shared.nempty);       /* wait for at least
  1 empty slot */
      sem_wait(&shared.mutex);

      shared.buff[i % NBUFF] = i;     /* store i into
  circular buffer */
  #ifdef DEBUG
      printf("wrote %d to buffer at location %d\n", i, i %
  NBUFF);
  #endif

      sem_post(&shared.mutex);
      sem_post(&shared.nstored);      /* 1 more stored
  item */
    }

    return (NULL);
  }
  /* end producer */
  ```
  ```

- Meanwhile, the consumer thread will enter the loop and wait on the semaphore *nstored*.
- Since initially *nstored* is set to 0, this call will block and the consumer will wait until the producer posts on the semaphore *nstored*.
- When the producer posts on the semaphore *nstored*, the consumer will return from *sem_wait* on *nstored* and invoke the *sem_wait* on the semaphore *mutex*.
- If the producer is not in the critical section, the consumer will obtain the *mutex* semaphore, consume the buffer (we simply check if the value in the buffer match the corresponding *(loop index mod NBUFF)* and print an error message in case they don't match), and release the mutex by calling *sem_post* on the *mutex* semaphore.
- Then the consumer thread will post the *sem_post* on the semaphore *nempty* to indicate to the producer that now there is an empty slot. The consumer thread terminates when the loop completes (i.e., after *nitems* iterations).

- ```
  /* consumer function */
  void *consumer(void *arg) {
    int i;

    for (i = 0; i < nitems; i++) {
      sem_wait(&shared.nstored);       /* wait for at
  least 1 stored item */
      sem_wait(&shared.mutex);

      if (shared.buff[i % NBUFF] != i)
        printf("error: buff[%d] = %d\n", i, shared.buff[i
  % NBUFF]);
  #ifdef DEBUG
      printf("read %d from buffer at location %d\n",
              shared.buff[i % NBUFF], i % NBUFF);
  #endif

      sem_post(&shared.mutex);
      sem_post(&shared.nempty);        /* 1 more empty
  slot */
    }

    return (NULL);
  }
  /* end consumer *
  ```
  ```

- You can compile the program with the DEBUG variable defined using -DDEBUG during compilation and see how the two threads progress. Here is a sample output when we execute the program with 20 items. You will notice that the output would be different every time you execute the program even with the same number of elements.

- **gcc -O -Wall -DDEBUG prodcons1.c -lpthread**

```
$ ./a.out 20
wrote 0 to buffer at location 0
wrote 1 to buffer at location 1
wrote 2 to buffer at location 2
wrote 3 to buffer at location 3
wrote 4 to buffer at location 4
wrote 5 to buffer at location 5
wrote 6 to buffer at location 6
wrote 7 to buffer at location 7
wrote 8 to buffer at location 8
read 0 from buffer at location 0
read 1 from buffer at location 1
read 2 from buffer at location 2
read 3 from buffer at location 3
read 4 from buffer at location 4
read 5 from buffer at location 5
read 6 from buffer at location 6
read 7 from buffer at location 7
read 8 from buffer at location 8
wrote 9 to buffer at location 9
wrote 10 to buffer at location 0
wrote 11 to buffer at location 1
wrote 12 to buffer at location 2
wrote 13 to buffer at location 3
wrote 14 to buffer at location 4
wrote 15 to buffer at location 5
wrote 16 to buffer at location 6
wrote 17 to buffer at location 7
wrote 18 to buffer at location 8
read 9 from buffer at location 9
read 10 from buffer at location 0
read 11 from buffer at location 1
read 12 from buffer at location 2
read 13 from buffer at location 3
read 14 from buffer at location 4
read 15 from buffer at location 5
read 16 from buffer at location 6
read 17 from buffer at location 7
read 18 from buffer at location 8
wrote 19 to buffer at location 9
read 19 from buffer at location 9
```

prodcons2.c

- Solution to multiple producer and single consumer problem:

```c
/* Solution to the Multiple Producer/Single Consumer problem using
   semaphores. This example uses a circular buffer to put and get the
   data (a bounded-buffer).
   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens

   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
*/

/* include globals */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>     /* for POSIX threads */
#include <semaphore.h>   /* for POSIX semaphores */

#define  min(a, b)  (((a) < (b)) ? (a) : (b))

#define  NBUFF          10
#define  MAXNTHREADS    100
```

```c

int      nitems, nproducers;                      /* read-only */

struct {                    /* data shared by producers and consumers */
  int       buff[NBUFF];
  int       nput;                            /* item number: 0, 1, 2, ... */
  int       nputval;                         /* value to store in buff[] */
  sem_t     mutex, nempty, nstored;          /* semaphores, not pointers */
} shared;

void       *producer(void *), *consumer(void *);

/* end globals */

/* main program */
int main(int argc, char **argv)
{
  int             i, prodcount[MAXNTHREADS];
  pthread_t       tid_producer[MAXNTHREADS], tid_consumer;

  if (argc != 3) {
    printf("Usage: %s <#items> <#producers>\n", argv[0]);
    exit(-1);
  }

```

```c
  nitems = atoi(argv[1]);
  nproducers = min(atoi(argv[2]), MAXNTHREADS);

  /* initialize three semaphores */
  sem_init(&shared.mutex, 0, 1);
  sem_init(&shared.nempty, 0, NBUFF);
  sem_init(&shared.nstored, 0, 0);

  /* create all producers and one consumer */
  for (i = 0; i < nproducers; i++) {
    prodcount[i] = 0;
    pthread_create(&tid_producer[i], NULL, producer, &prodcount[i]);
  }
  pthread_create(&tid_consumer, NULL, consumer, NULL);

  /* wait for all producers and the consumer */
  for (i = 0; i < nproducers; i++) {
    pthread_join(tid_producer[i], NULL);
    printf("producer count[%d] = %d\n", i, prodcount[i]);
  }
  pthread_join(tid_consumer, NULL);

  sem_destroy(&shared.mutex);
  sem_destroy(&shared.nempty);
  sem_destroy(&shared.nstored);

  return 0;
}
/* end main */
```

```c
/* producer function */
void *producer(void *arg)
{
  for ( ; ; ) {
    sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
    sem_wait(&shared.mutex);

    if (shared.nput >= nitems) {
      sem_post(&shared.nempty);
      sem_post(&shared.mutex);
      return(NULL);                 /* all done */
    }

    shared.buff[shared.nput % NBUFF] = shared.nputval;
#ifdef DEBUG
    printf("wrote %d to buffer at location %d\n",
           shared.nputval, shared.nput % NBUFF);
#endif
    shared.nput++;
    shared.nputval++;

    sem_post(&shared.mutex);
    sem_post(&shared.nstored);      /* 1 more stored item */
    *((int *) arg) += 1;
  }
}
/* end producer */
```

```c

/* consumer function */
void *consumer(void *arg)
{
  int i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nstored);        /* wait for at least 1 stored item */
    sem_wait(&shared.mutex);

    if (shared.buff[i % NBUFF] != i)
      printf("error: buff[%d] = %d\n", i, shared.buff[i % NBUFF]);
#ifdef DEBUG
    printf("read %d from buffer at location %d\n",
           shared.buff[i % NBUFF], i % NBUFF);
#endif

    sem_post(&shared.mutex);
    sem_post(&shared.nempty);         /* 1 more empty slot */
  }

  return (NULL);
}
/* end consumer */
```

prodcons3.c

- Solution to multiple producer and multiple consumer problem:

```c
/* Solution to the Multiple Producer/Multiple Consumer problem using
   semaphores. This example uses a circular buffer to put and get the
   data (a bounded-buffer).
   Source: UNIX Network Programming, Volume 2 by W. Richard Stevens

   To compile: gcc -O -Wall -o <filename> <filename>.c -lpthread
*/

/* include globals */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>     /* for POSIX threads */
#include <semaphore.h>   /* for POSIX semaphores */

#define  min(a, b)  (((a) < (b)) ? (a) : (b))
```

```c

#define  NBUFF          10
#define  MAXNTHREADS    100

int      nitems, nproducers, nconsumers;              /* read-only */

struct {                  /* data shared by producers and consumers */
  int      buff[NBUFF];
  int      nput;                          /* item number: 0, 1, 2, ... */
  int      nputval;                       /* value to store in buff[] */
  int      nget;                          /* item number: 0, 1, 2, ... */
  int      ngetval;                       /* value fetched from buff[] */
  sem_t    mutex, nempty, nstored;        /* semaphores, not pointers */
} shared;

void       *producer(void *), *consumer(void *);

/* end globals */

/* main program */
int main(int argc, char **argv)
{
  int              i, prodcount[MAXNTHREADS], conscount[MAXNTHREADS];
  pthread_t        tid_producer[MAXNTHREADS], tid_consumer[MAXNTHREADS];

  if (argc != 4) {
    printf("Usage: %s <#items> <#producers> <#consumers>\n", argv[0]);
    exit(-1);
  }
```

```c

    nitems = atoi(argv[1]);
    nproducers = min(atoi(argv[2]), MAXNTHREADS);
    nconsumers = min(atoi(argv[3]), MAXNTHREADS);

    /* initialize three semaphores */
    sem_init(&shared.mutex, 0, 1);
    sem_init(&shared.nempty, 0, NBUFF);
    sem_init(&shared.nstored, 0, 0);

    /* create all producers and all consumers */
    for (i = 0; i < nproducers; i++) {
      prodcount[i] = 0;
      pthread_create(&tid_producer[i], NULL, producer, &prodcount[i]);
    }
    for (i = 0; i < nconsumers; i++) {
      conscount[i] = 0;
      pthread_create(&tid_consumer[i], NULL, consumer, &conscount[i]);
    }

    /* wait for all producers and all consumers */
    for (i = 0; i < nproducers; i++) {
      pthread_join(tid_producer[i], NULL);
      printf("producer count[%d] = %d\n", i, prodcount[i]);
    }
    for (i = 0; i < nconsumers; i++) {
      pthread_join(tid_consumer[i], NULL);
      printf("consumer count[%d] = %d\n", i, conscount[i]);
    }
```

```c

  sem_destroy(&shared.mutex);
  sem_destroy(&shared.nempty);
  sem_destroy(&shared.nstored);

  return 0;
}
/* end main */

/* producer function */
void *producer(void *arg)
{
  for ( ; ; ) {
    sem_wait(&shared.nempty);       /* wait for at least 1 empty slot */
    sem_wait(&shared.mutex);

    if (shared.nput >= nitems) {
      sem_post(&shared.nstored);    /* let consumers terminate */
      sem_post(&shared.nempty);
      sem_post(&shared.mutex);
      return(NULL);                 /* all done */
    }

    shared.buff[shared.nput % NBUFF] = shared.nputval;
    shared.nput++;
    shared.nputval++;

    sem_post(&shared.mutex);
    sem_post(&shared.nstored);      /* 1 more stored item */
    *((int *) arg) += 1;
  }
}
/* end producer */
```

```c

/* consumer function */
void *consumer(void *arg)
{
  int i;

  for ( ; ; ) {
    sem_wait(&shared.nstored);        /* wait for at least 1 stored item */
    sem_wait(&shared.mutex);

    if (shared.nget >= nitems) {
      sem_post(&shared.nstored);
      sem_post(&shared.mutex);
      return(NULL);                   /* all done */
    }

    i = shared.nget % NBUFF;
    if (shared.buff[i] != shared.ngetval)
      printf("error: buff[%d] = %d\n", i, shared.buff[i]);
    shared.nget++;
    shared.ngetval++;

    sem_post(&shared.mutex);
    sem_post(&shared.nempty);         /* 1 more empty slot */
    *((int *) arg) += 1;
  }
}
/* end consumer */

```

Semaphore for Resource Allocation

- Pool of N problems
- Resource sharing among multiple processes / uninterrupted period
- Limit the highest number of resources in use at any time
- ………

Frame	A fixed-length block of main memory.
Page	A fixed-length block of data that resides in secondary memory (such as disk). A page of data may temporarily be copied into a frame of main memory.
Segment	A variable-length block of data that resides in secondary memory. An entire segment may temporarily be copied into an available region of main memory (segmentation) or the segment may be divided into pages which can be individually copied into main memory (combined segmentation and paging).

Table 7.1

Memory Management Terms

Memory Management Requirements

- Relocation
- Protection
- Sharing
- Logical organization
- Physical organization

Relocation

- Programmers typically do not know in advance which other programs will be resident in main memory at the time of execution of their program
- Active processes need to be able to be swapped in and out of main memory in order to maximize processor utilization
- Specifying that a process must be placed in the same memory region when it is swapped back in would be limiting
  - May need to *relocate* the process to a different area of memory

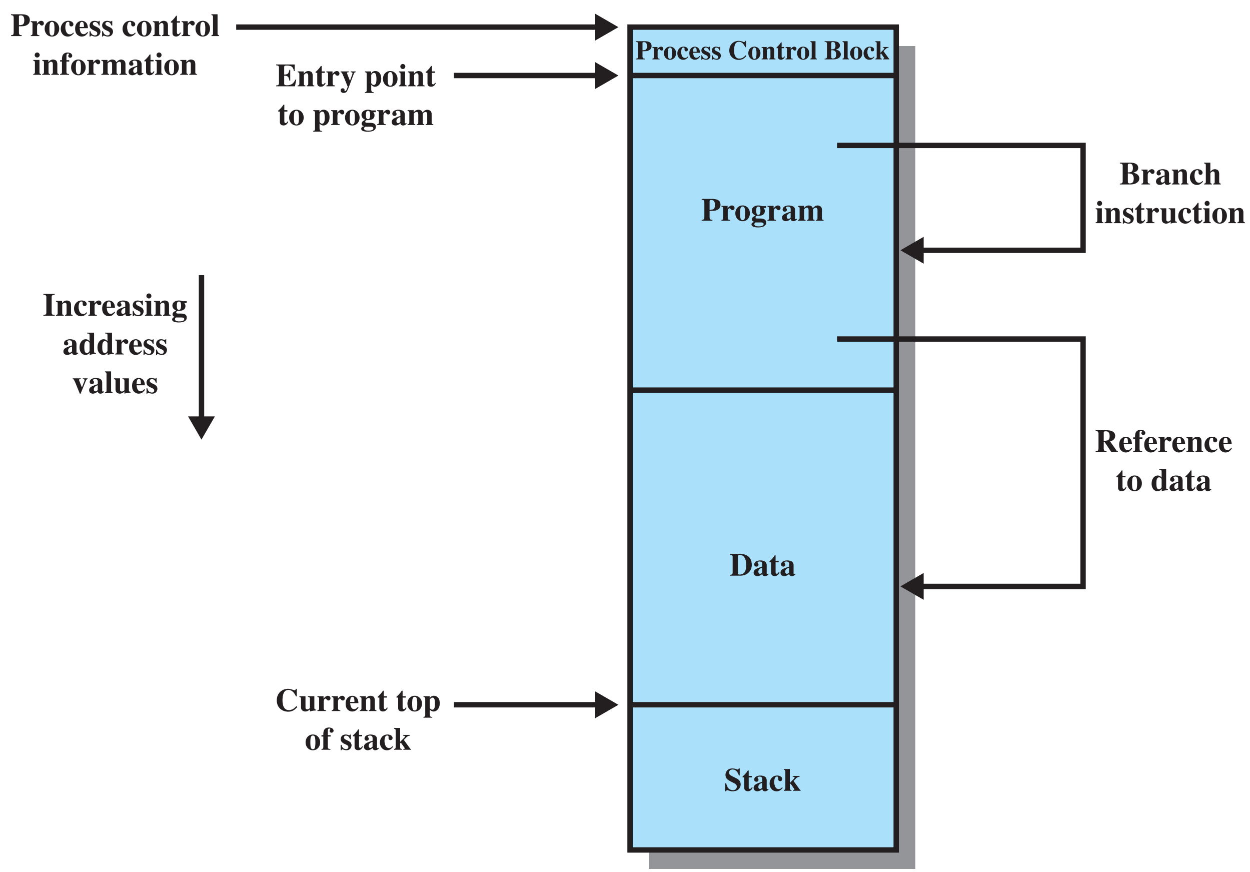


Figure 7.1 Addressing Requirements for a Process

Protection

- Processes need to acquire permission to reference memory locations for reading or writing purposes
- Location of a program in main memory is unpredictable
- Memory references generated by a process must be checked at run time
- Mechanisms that support relocation also support protection

Sharing

- Advantageous to allow each process access to the same copy of the program rather than have their own separate copy
- Memory management must allow controlled access to shared areas of memory without compromising protection
- Mechanisms used to support relocation support sharing capabilities

Logical Organization

- Memory is organized as linear

Programs are written in modules

- Modules can be written and compiled independently
- Different degrees of protection given to modules (read-only, execute-only)
- Sharing on a module level corresponds to the user’s way of viewing the problem

- Segmentation is the tool that most readily satisfies requirements

Physical Organization

- Computer memory is organized into at least two levels, referred to as main memory and secondary memory
- Cannot leave the programmer with the responsibility to manage memory, this is impractical and undesirable for two reasons:
  - Memory available for a program plus its data may be insufficient
    - Overlaying allows various modules to be assigned the same region of memory but is time consuming to program
  - Programmer does not know how much space will be available

Memory Partitioning

- Memory management brings processes into main memory for execution by the processor
  - Involves virtual memory
  - Based on segmentation and paging
- Partitioning
  - Used in several variations in some now-obsolete operating systems
  - Does not involve virtual memory

Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation.
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.

Table 7.2

Memory Management Techniques

(Table is on page 317 in textbook)

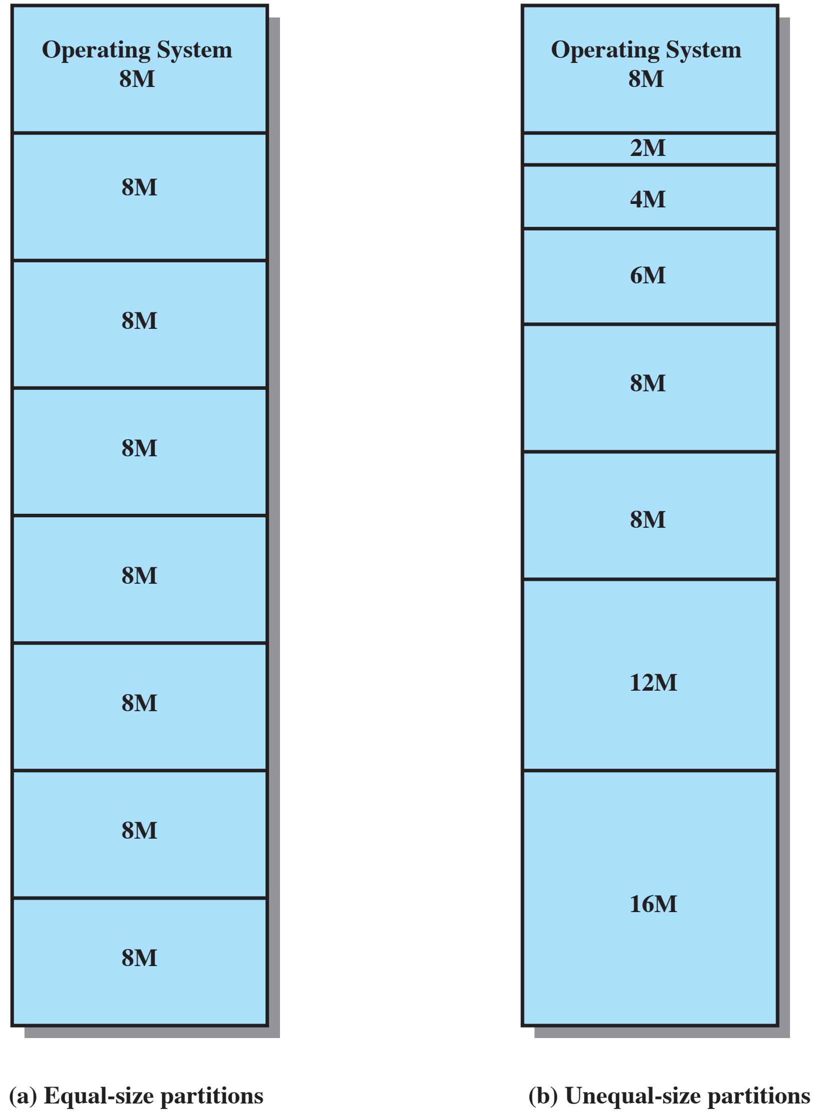
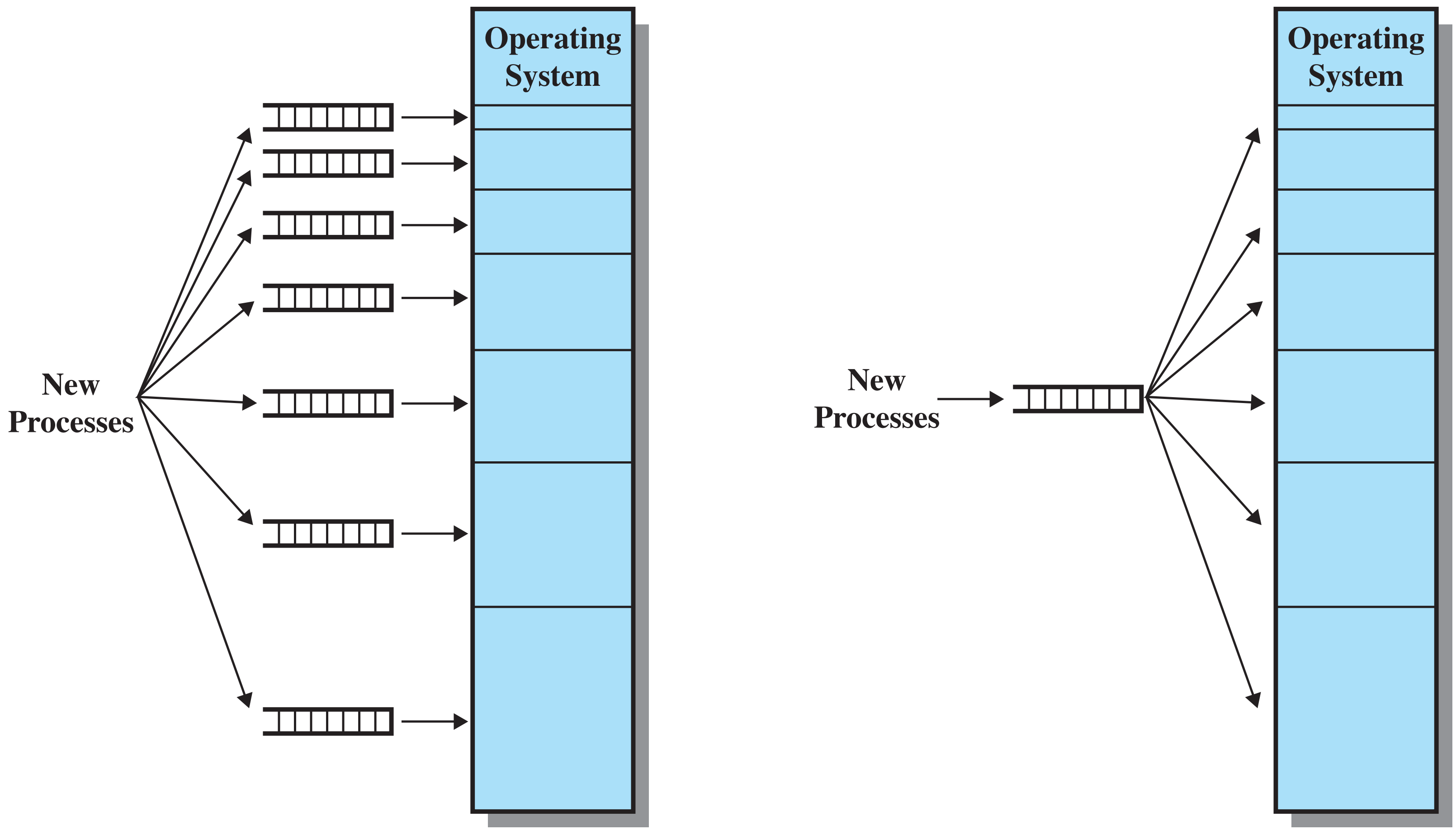


Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

Disadvantages

- A program may be too big to fit in a partition
    - Program needs to be designed with the use of overlays
- Main memory utilization is inefficient
    - Any program, regardless of size, occupies an entire partition
    - ***Internal fragmentation* –** Wasted space due to the block of data loaded being smaller than the partition



(a) One process queue per partition

(b) Single queue

Figure 7.3 Memory Assignment for Fixed Partitioning

Disadvantages

- The number of partitions specified at system generation time limits the number of active processes in the system
- Small jobs will not utilize partition space efficiently

Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as it requires
- This technique was used by IBM's mainframe operating system, OS/MVT (Multiprogramming with a Variable Number of Tasks)

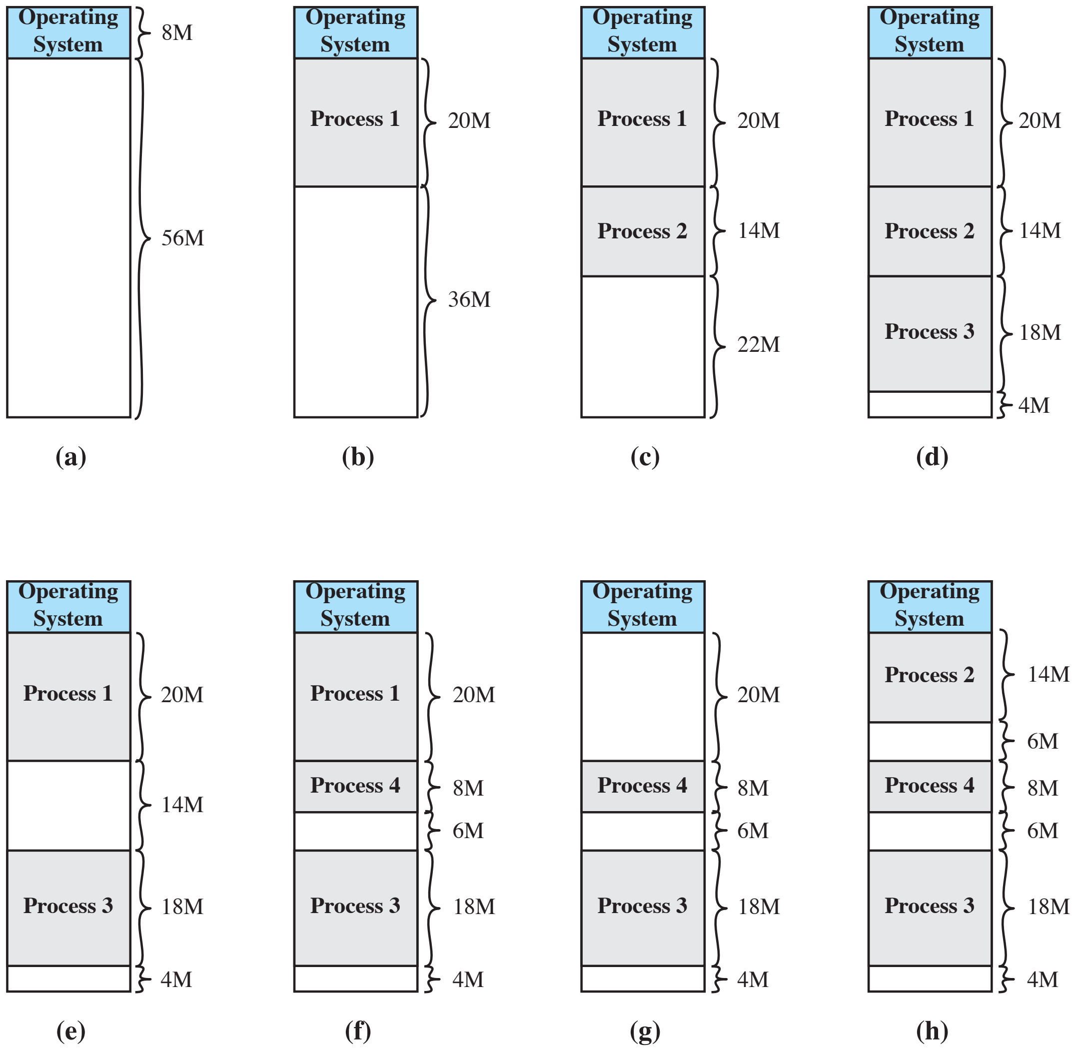


Figure 7.4 The Effect of Dynamic Partitioning

Dynamic Partitioning

External Fragmentation

- Memory becomes more and more fragmented
- Memory utilization declines

Compaction

- Technique for overcoming external fragmentation
- OS shifts processes so that they are contiguous
- Free memory is together in one block
- Time consuming and wastes CPU time

Placement Algorithms

Best-fit	First-fit	Next-fit
• Chooses the block that is closest in size to the request	• Begins to scan memory from the beginning and chooses the first available block that is large enough	• Begins to scan memory from the location of the last placement and chooses the next available block that is large enough

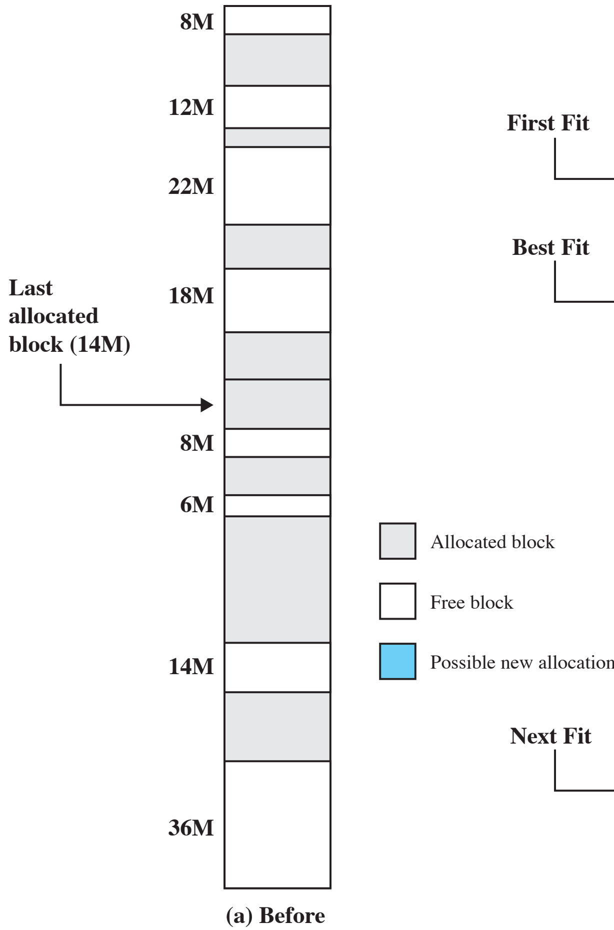


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

Buddy System

- A reasonable compromise to overcome the disadvantages of both the fixed and variable partitioning schemes by combining fixed and dynamic partitioning schemes
- Space available for allocation is treated as a single block
- Memory blocks are available of size 2^K *words,* $L \leq K \leq U$, where
    - 2^L = smallest size block that is allocated
    - 2^U = largest size block that is allocated; generally 2^U is the size of the entire memory available for allocation

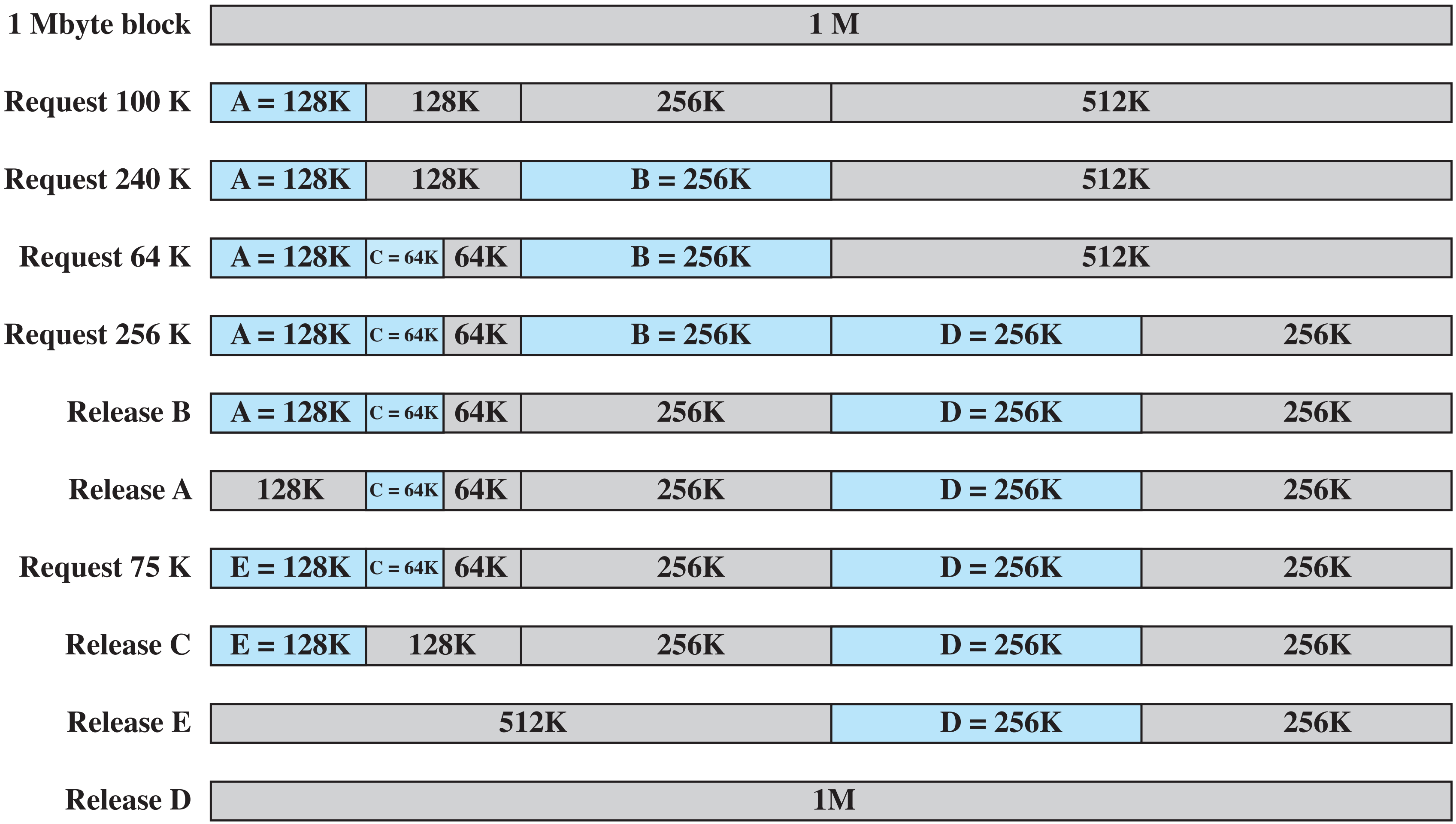


Figure 7.6 Example of Buddy System

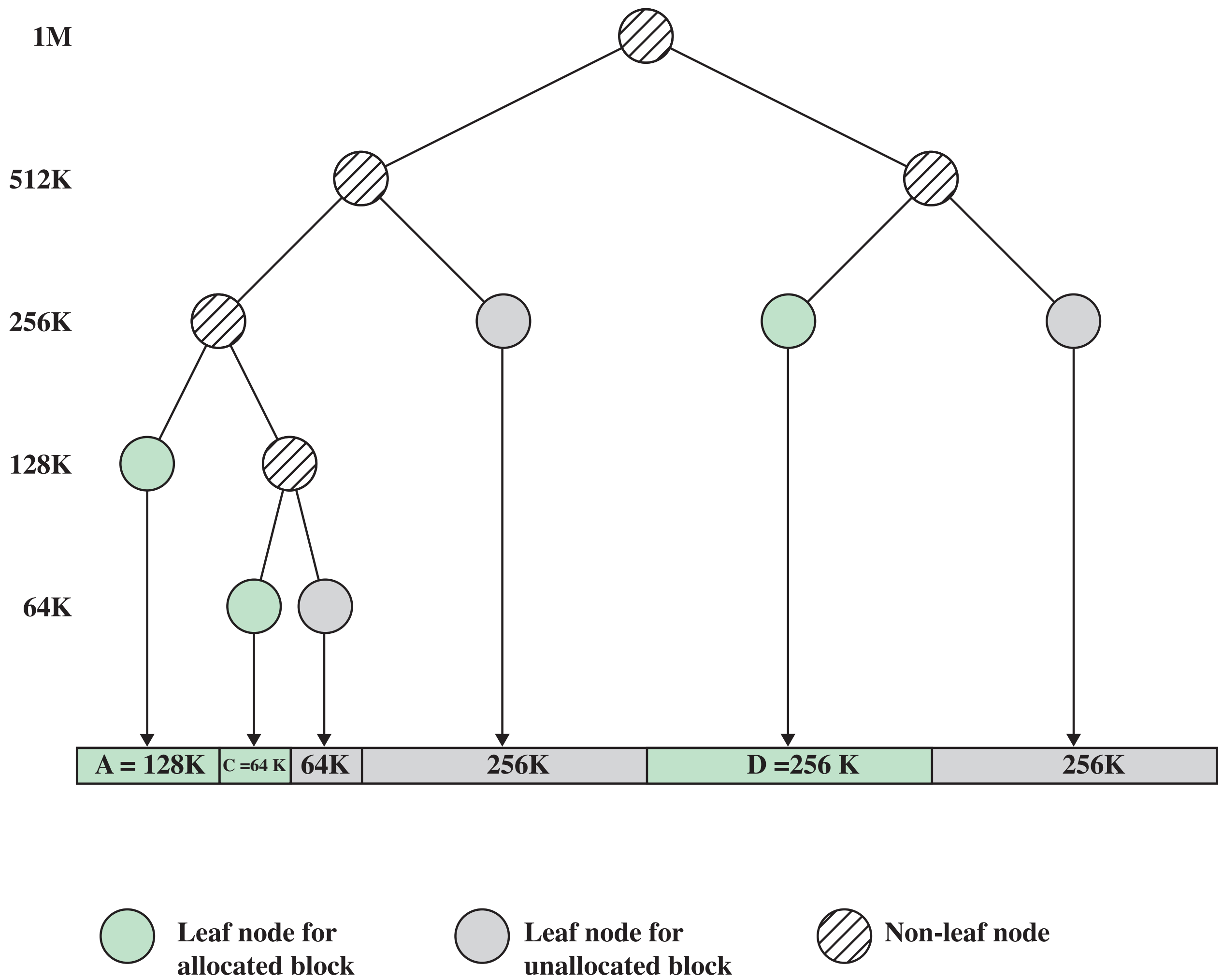


Figure 7.7 Tree Representation of Buddy System

Relocation, I.

- When the fixed partition scheme is used, we can expect a process will always be assigned to the same partition
  - Whichever partition is selected when a new process is loaded will always be used to swap that process back into memory after it has been swapped out
  - In that case, a simple relocating loader can be used
  - When the process is first loaded, all relative memory references in the code are replaced by absolute main memory addresses, determined by the base address of the loaded process

Relocation, II.

- In the case of equal-size partitions and in the case of a single process queue for unequal-size partitions, a process may occupy different partitions during the course of its life
  - When a process image is first created, it is loaded into some partition in main memory; Later, the process may be swapped out
  - When it is subsequently swapped back in, it may be assigned to a different partition than the last time
  - The same is true for dynamic partitioning
- When compaction is used, processes are shifted while they are in main memory
  - Thus, the locations referenced by a process are not fixed
  - They will change each time a process is swapped in or shifted

Addresses

Logical

- Reference to a memory location independent of the current assignment of data to memory

Relative

- A particular example of logical address, in which the address is expressed as a location relative to some known point

Physical or Absolute

- Actual location in main memory

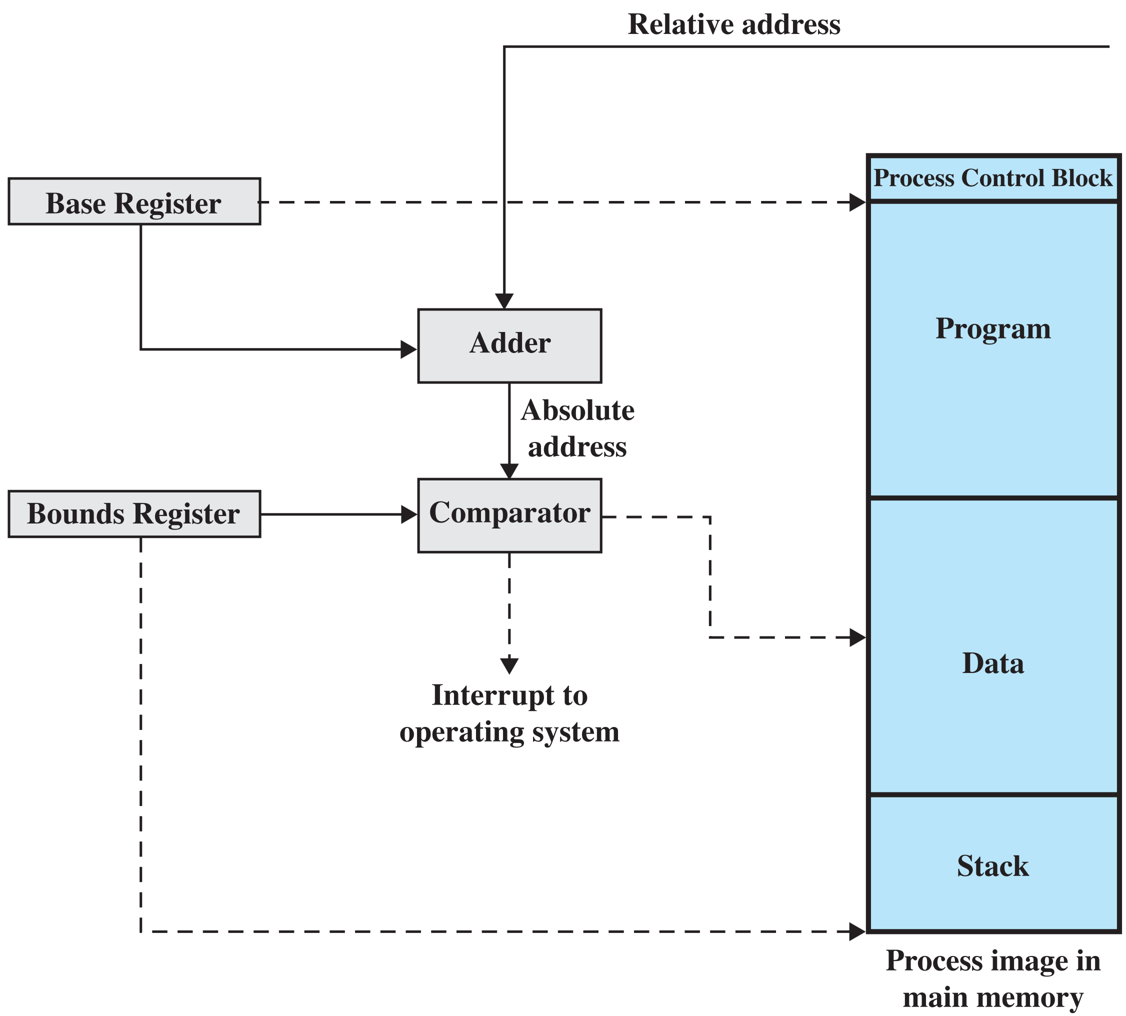


Figure 7.8 Hardware Support for Relocation

Paging

- Partition memory into equal fixed-size chunks that are relatively small
- Process is also divided into small fixed-size chunks of the same size

Pages	Frames
• Chunks of a process	• Available chunks of memory

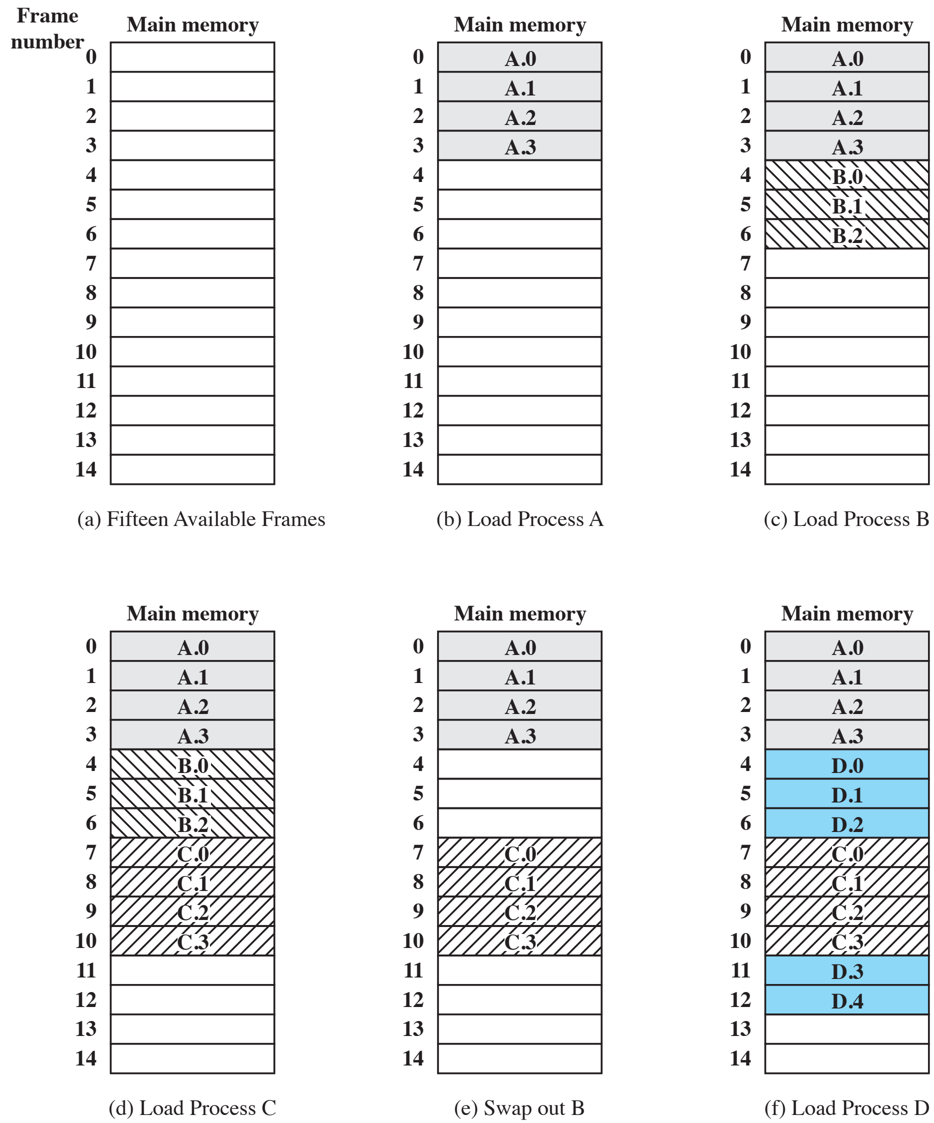


Figure 7.9 Assignment of Process Pages to Free Frames

Page Table

- Maintained by operating system for each process
- Contains the frame location for each page in the process
- Processor must know how to access for the current process
- Used by processor to produce a physical address

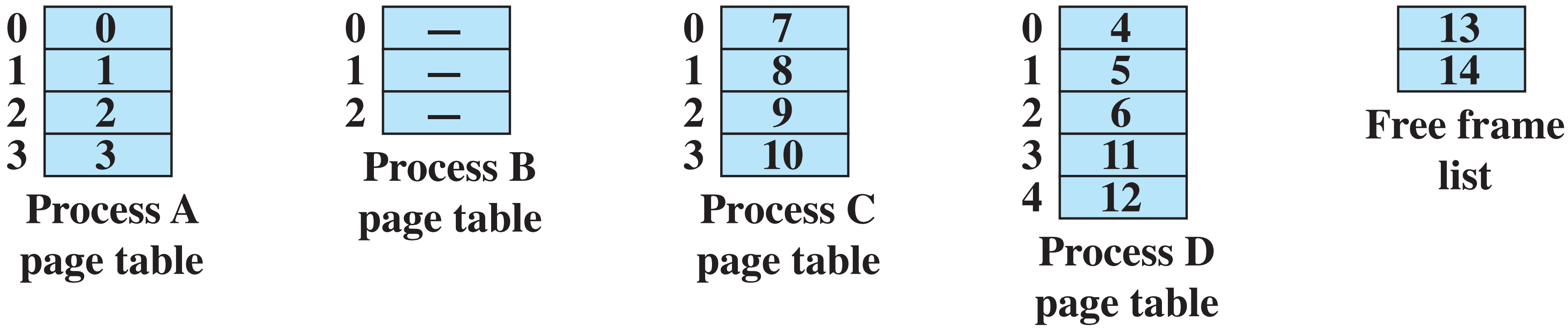


Figure 7.10 Data Structures for the Example of Figure 7.9 at Time Epoch (f)

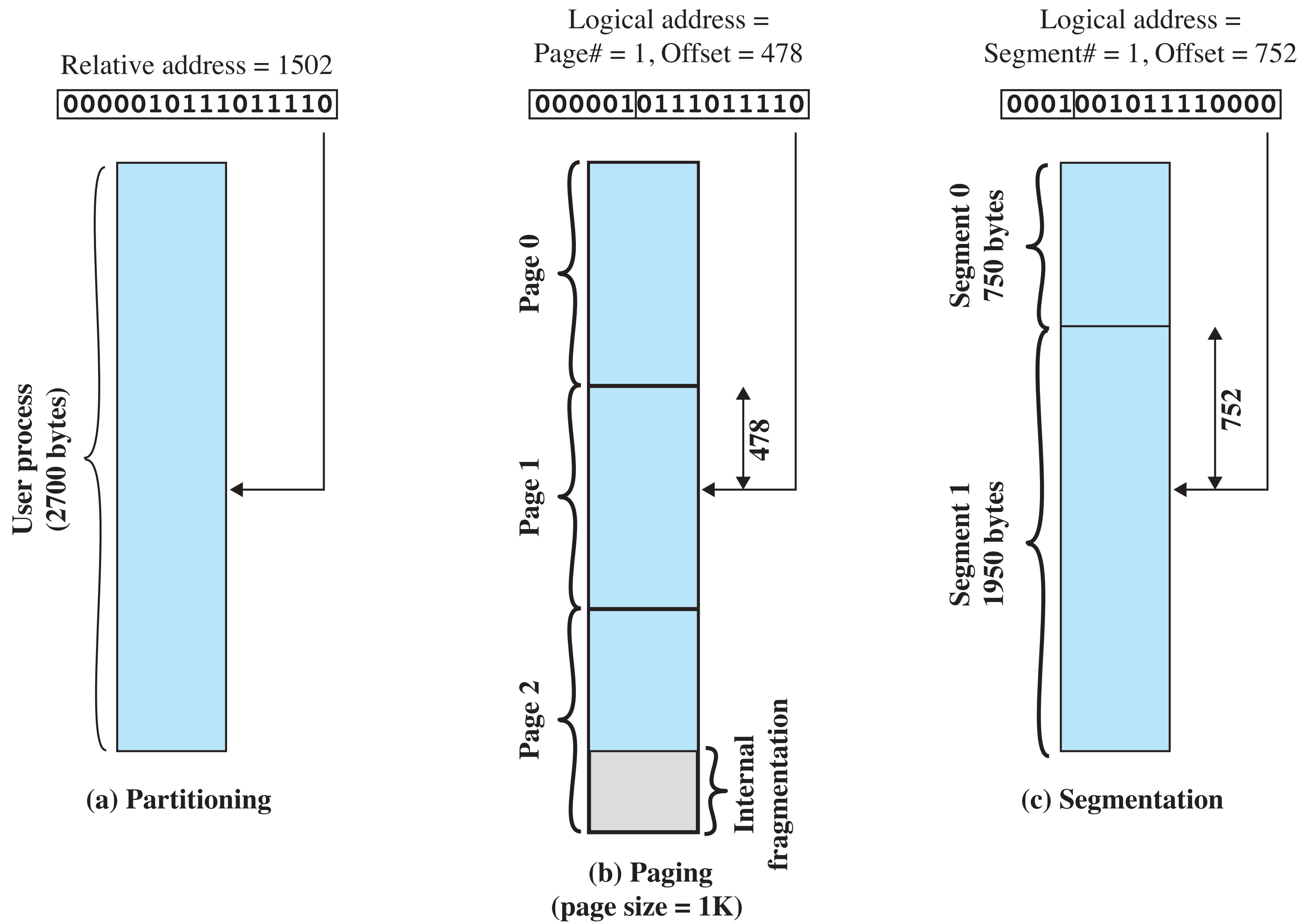


Figure 7.11 Logical Addresses

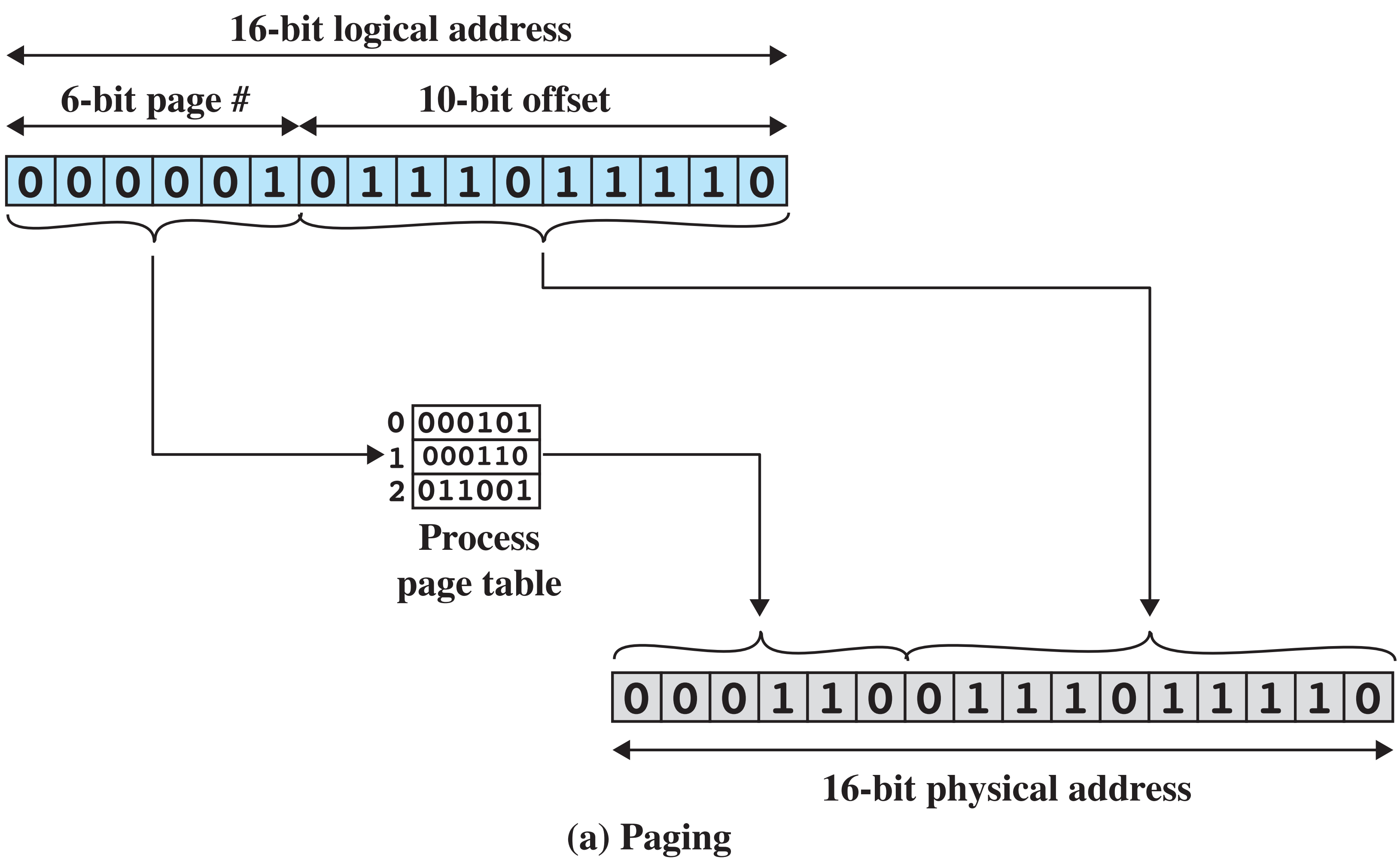


Figure 7.12 Examples of Logical-to-Physical Address Translation

Segmentation

- A program can be subdivided into segments
  - May vary in length
  - There is a maximum length
- Addressing consists of two parts:
  - Segment number
  - An offset
- Similar to dynamic partitioning
- Eliminates internal fragmentation

Segmentation

- Usually visible
- Provided as a convenience for organizing programs and data
- Typically the programmer will assign programs and data to different segments
- For purposes of modular programming the program or data may be further broken down into multiple segments
  - The principal inconvenience of this service is that the programmer must be aware of the maximum segment size limitation

Address Translation

- Another consequence of unequal size segments is that there is no simple relationship between logical addresses and physical addresses
- The following steps are needed for address translation:
  - Extract the segment number as the leftmost *n* bits of the logical address
  - Use the segment number as an index into the process segment table to find the starting physical address of the segment
  - Compare the offset, expressed in the rightmost *m* bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid
  - The desired physical address is the sum of the starting physical address of the segment plus the offset

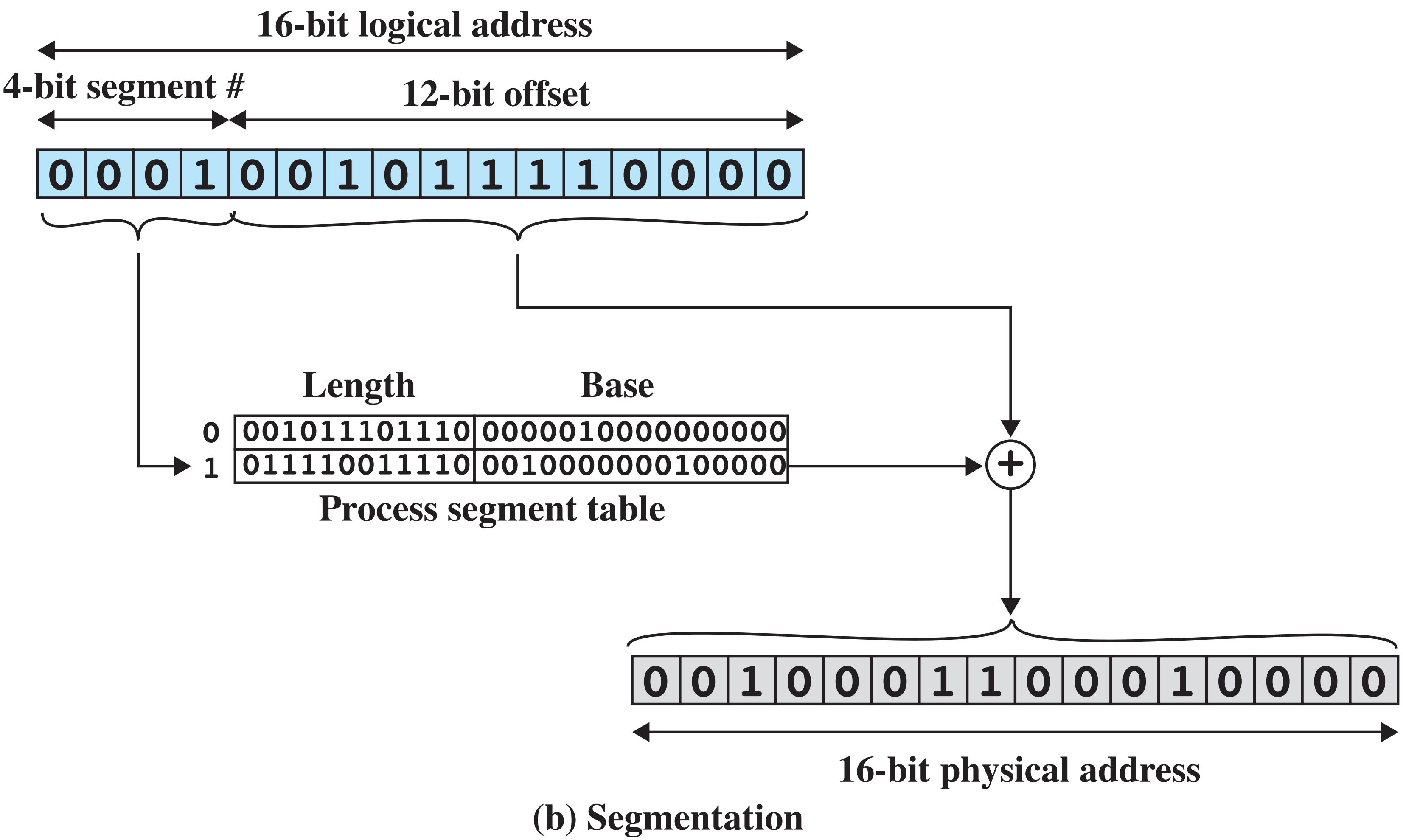


Figure 7.12 Examples of Logical-to-Physical Address Translation

Summary

- Memory management requirements
  - Relocation
  - Protection
  - Sharing
  - Logical organization
  - Physical organization
- Paging
- Memory partitioning
  - Fixed partitioning
  - Dynamic partitioning
  - Buddy system
  - Relocation
- Segmentation