5.7 Classical Problems of Synchronization

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

Bounded-Buffer Problem

- ***BUF_SZ*** elements inside the shared buffer
- Semaphore **`num_full_el`** initialized to the value 0 – keeps track of the number of elements that are full.
- Semaphore **`num_empty_el`** initialized to the value n – keeps track of number of elements that are empty.

- Why use semaphores, when the previous approach seemed to work fine (i.e. using the in and out indices without using any synchronization primitives)?

Bounded-Buffer Problem

- ***BUF_SZ*** elements inside the shared buffer
- Semaphore **`num_full_el`** initialized to the value 0 – keeps track of the number of elements that are full.
- Semaphore **`num_empty_el`** initialized to the value n – keeps track of number of elements that are empty.

- Why use semaphores, when the previous approach seemed to work fine (i.e. using the in and out indices without using any synchronization primitives)?
  - Because of the busy-waiting problem in which a process or thread may be spending valuable CPU time doing nothing but waiting in a loop!
  - We may still use the in and out variables to index a particular buffer in the pool.

Bounded Buffer Problem (Cont.)

- The structure of the **producer** process

```
do {
   /* produce an item in next_produced
  */
   ...
   /* dec empty sem. */
   wait(num_empty_el);
   /* write/produce to an entry*/
   buffer[in] = next_produced;
   in = (in + 1) % BUF_SZ;
   /* inc full sem. */
   signal(num_full_el);
} while (true);
```

- The structure of the **consumer** process

```
do {
   /* dec full sem. */
   wait(num_full_el);
   /* read/consume an entry */
   next_consumed = buffer[out];
   out = (out + 1) % BUF_SZ;
   /*inc empty sem.*/
   signal(num_empty_el);
          ...
   /* consume the item in next consumed */
   ...
} while (true);
```

Readers-Writers Problem

- A data set (e.g. a database) is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can read and write
- Problem:
  - Allow multiple readers to read at the same time
  - Only one writer can access the shared data at the same time (i.e. no other writers or other readers are allowed)
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1 (to protect access to `read_count`)
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
   wait(rw_mutex);
          ...
   /* writing is performed */
          ...
   signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

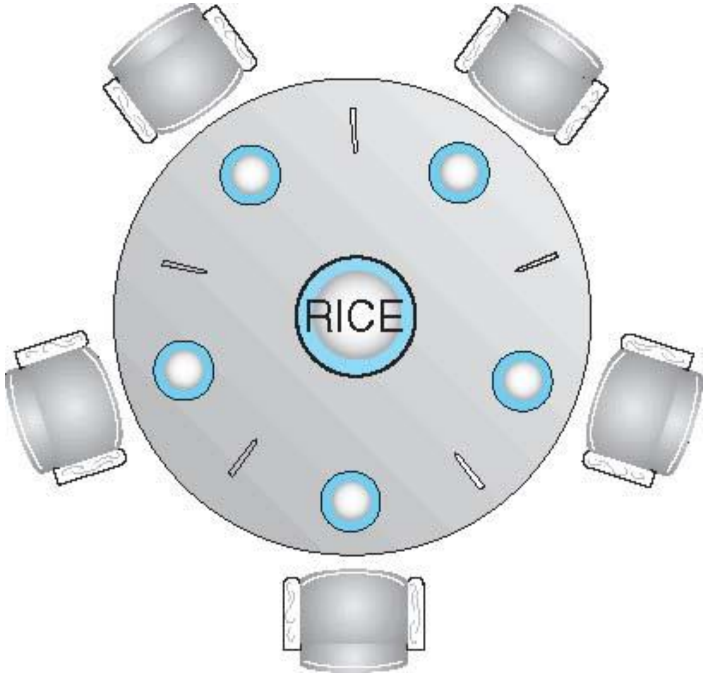
```
do {
     wait(mutex);
     read_count++;
     if (read_count == 1) /* only first reader locks rw_mutex */
           wait(rw_mutex);
     signal(mutex);
                ...
     /* reading dataset is performed, protected by rw_mutex */
     /* either one writer, or multiple readers at a time
                ...
     wait(mutex);
     read count--;
     if (read_count == 0) /* only last reader unlocks rw_mutex */
           signal(rw_mutex);
     signal(mutex);
 } while (true);
```

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP (i.e. no readers are allowed to read till after the writer gets and is done with his access)
- Both may have **starvation** leading to even more variations
- In some systems, the kernel provides a reader-writer locks

Dining-Philosophers Problem

- Philosophers spend their lives alternating between **thinking** and **eating**
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data analogy:
    - Each philosopher is a thread
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
     wait (chopstick[i] );
     wait (chopstick[ (i + 1) % 5] );

     //  eat

     signal (chopstick[i] );
     signal (chopstick[ (i + 1) % 5] );

     //  think

} while (TRUE);
```

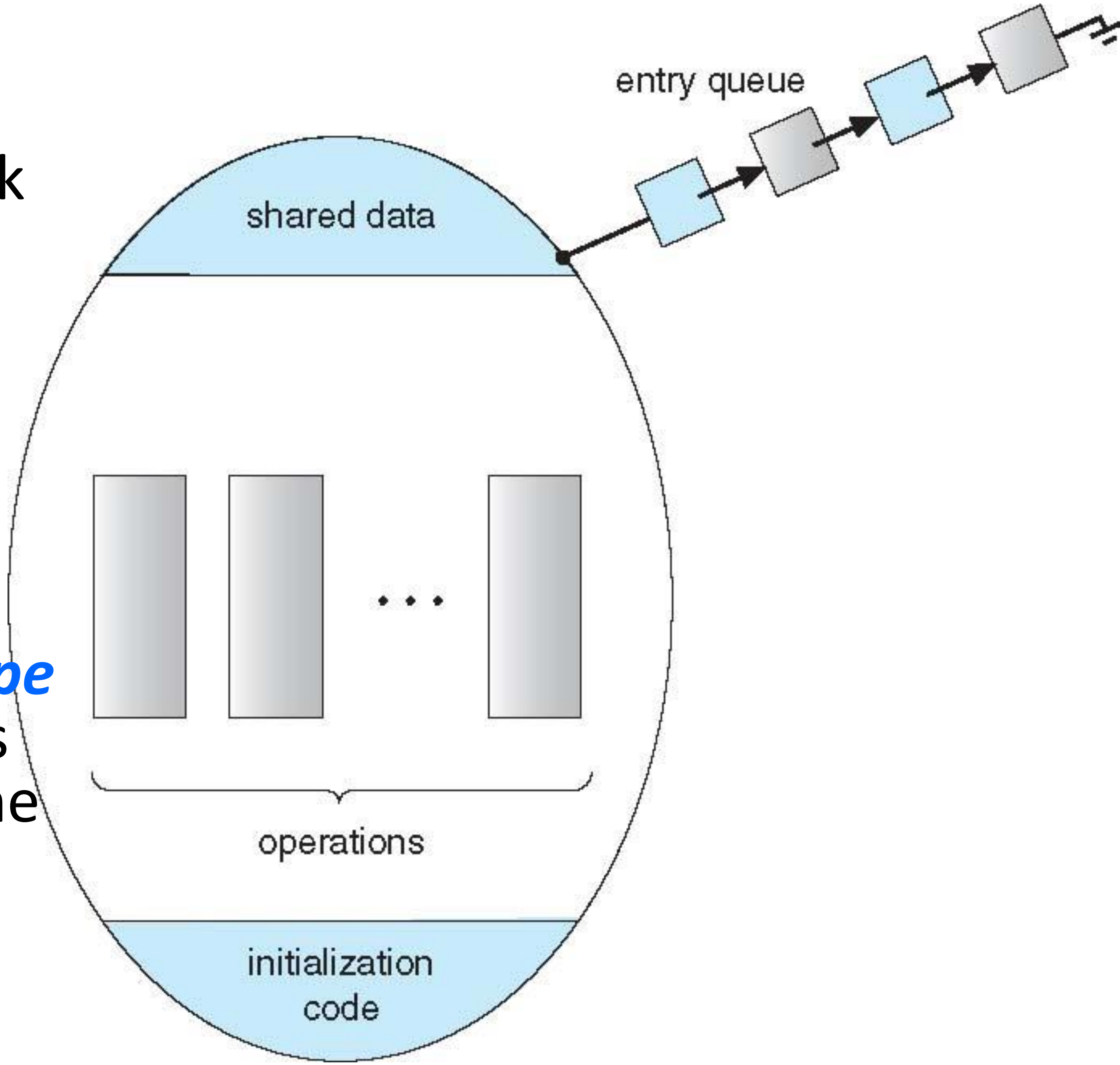
- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table of 5 chopsticks.
  - Use an asymmetric solution
    - An odd-numbered philosopher picks up the left chopstick first and then the right chopstick.
    - An even-numbered philosopher picks up the right chopstick first and then the left chopstick.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section) → we may use **monitors** to implement this method.

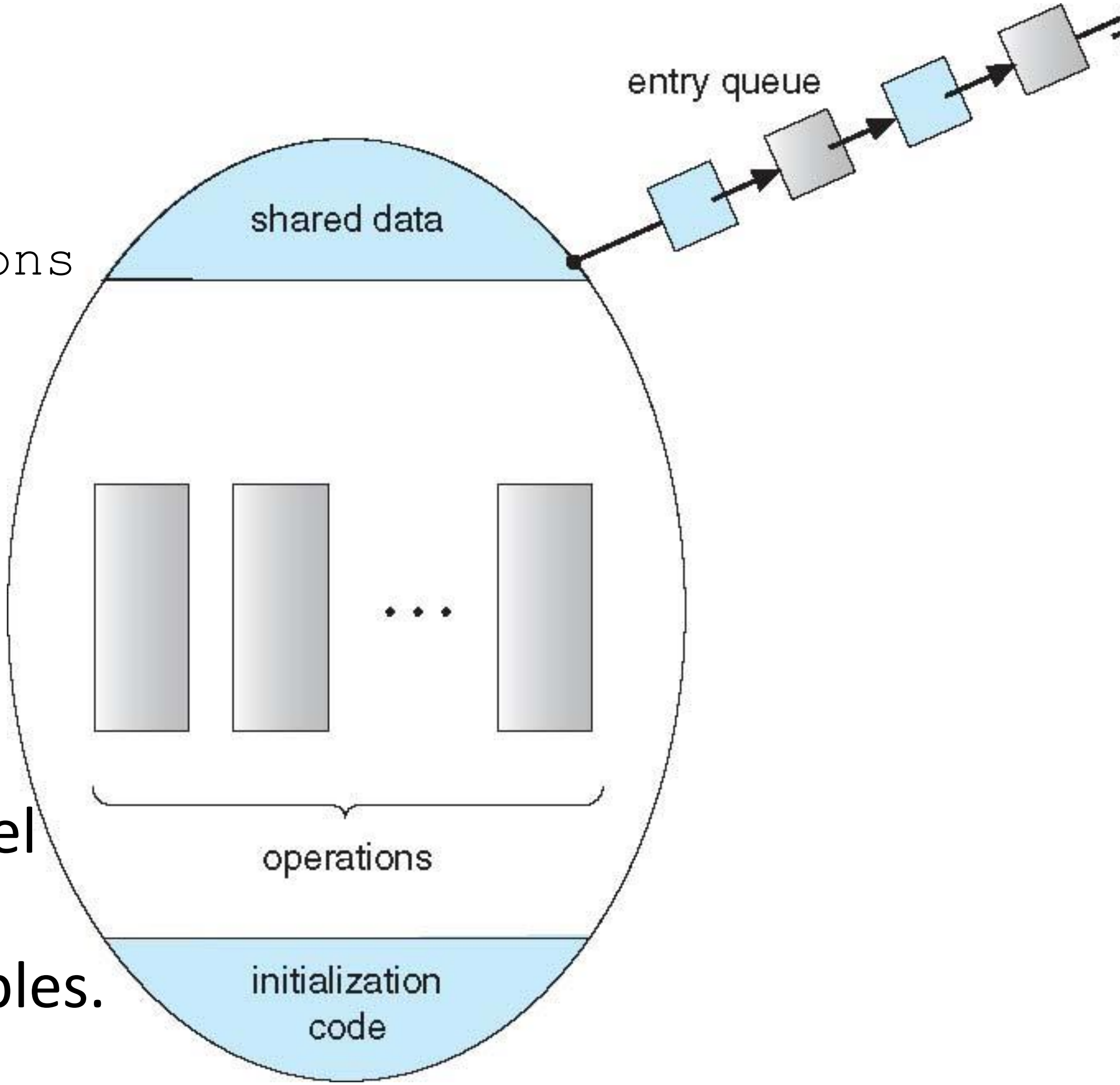
5.8 Monitors

- The dining philosophers deadlock may be solved using monitors.
- A monitor is a high-level abstraction that provides a convenient and effective mechanism for process synchronization
- A monitor is an ***abstract data type*** *(i.e. an **object**)*, internal variables only accessible by code within the procedure
- Only one process may be **active** within the monitor at a time.



5.8 Monitors

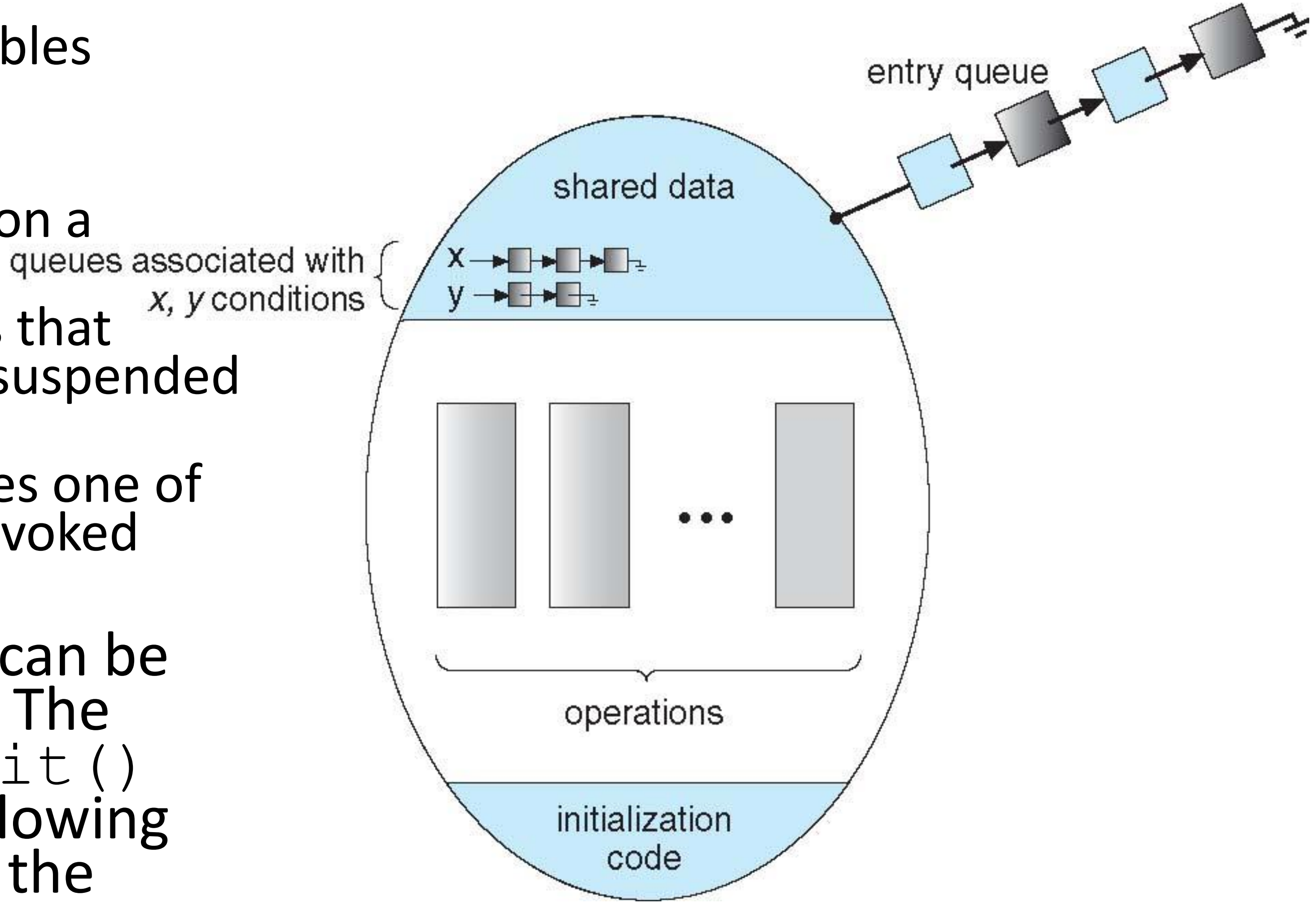
```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

 Initialization code (…) { … }

}
```



- But not powerful enough to model some synchronization schemes
- Thus we may use condition variables.

Condition Variables

- Condition variables are variables declared inside a monitor:
  `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
- Only one thread/process can be active inside the monitor. The process that issues `x.wait()` becomes inactive, thus allowing others to be active inside the monitor.



Condition Variables – cont.

- Contrast this operation with the `signal()` operation associated with semaphores, which always affects the state of the semaphore:
  - If there are no threads/processes that are waiting on the condition variable `x`, then calling `x.signal()` does not affect the condition variable.
  - Calling `signal(my_semaphore),` however, increments the semaphore whether there is someone waiting on it or not.

Condition Variables Choices

- If process P invokes `x.signal()`, and process Q was suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or Q blocks waiting on another condition (i.e. **immediate effect**).
  - **Signal and continue** – P continues till it leaves the monitor or blocks waiting on another condition, and then Q may become active (i.e. **deferred effect**).
  - Both have pros and cons – language implementer can decide

Monitor solution to dining philosophers

```
monitor DiningPhilosophers
{
  enum {THINKING, HUNGRY, EATING} state[5] ;
  condition cond[5];

  void pickup (int i) {
         state[i] = HUNGRY;
         test_and_signal(i); /* if successful: my state becomes EATING + signal
                                myself which is wasted cause I am not waiting*/
         if (state[i] != EATING) cond[i].wait(); /* test(i) did not succeed */
  }

  void putdown (int i) {
    state[i] = THINKING;
    // test left and right neighbors
    test_and_signal((i + 4) % 5);
    test_and_signal((i + 1) % 5);
  }
```

Monitor solution to Dining Philosophers (Cont.)

```
  void test_and_signal(int i) {
    if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ){
      state[i] = EATING;
      cond[i].signal();
    }
  }

  initialization_code() {
    for (int i = 0; i < 5; i++)
      state[i] = THINKING;
  }
} /* end of monitor */
```

Monitor solution to dining philosophers (Cont.)

- Each philosopher *i* invokes the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);
EAT
DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Monitor Implementation – Monitors without a condition variable

- Variables

  `semaphore mutex;  // (initially  = 1)`

- Each monitor procedure ***F*** will be replaced by

```
wait(mutex);
     …
     body of F;
     …
signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation – Monitors with a condition variables

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

```
semaphore x_sem; // (initially  = 0)
int x_count = 0; // num of processes suspended/waiting
                 // on condition variable
```

- We are implementing the **signal-and-wait** type condition variables.
  - A signaling process must wait until the signaled process either **leaves** or **blocks** waiting.
  - Thus we need another semaphore:

```
// signaling process uses next to suspend itself
semaphore next; // (initially  = 0)
int next_count = 0;  // num of processes suspended
                     // due to signal-and-wait
```

Monitor Implementation – Monitors with a condition variables (cont.)

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

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

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

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

Monitor Implementation – Monitors with a condition variables (cont.)

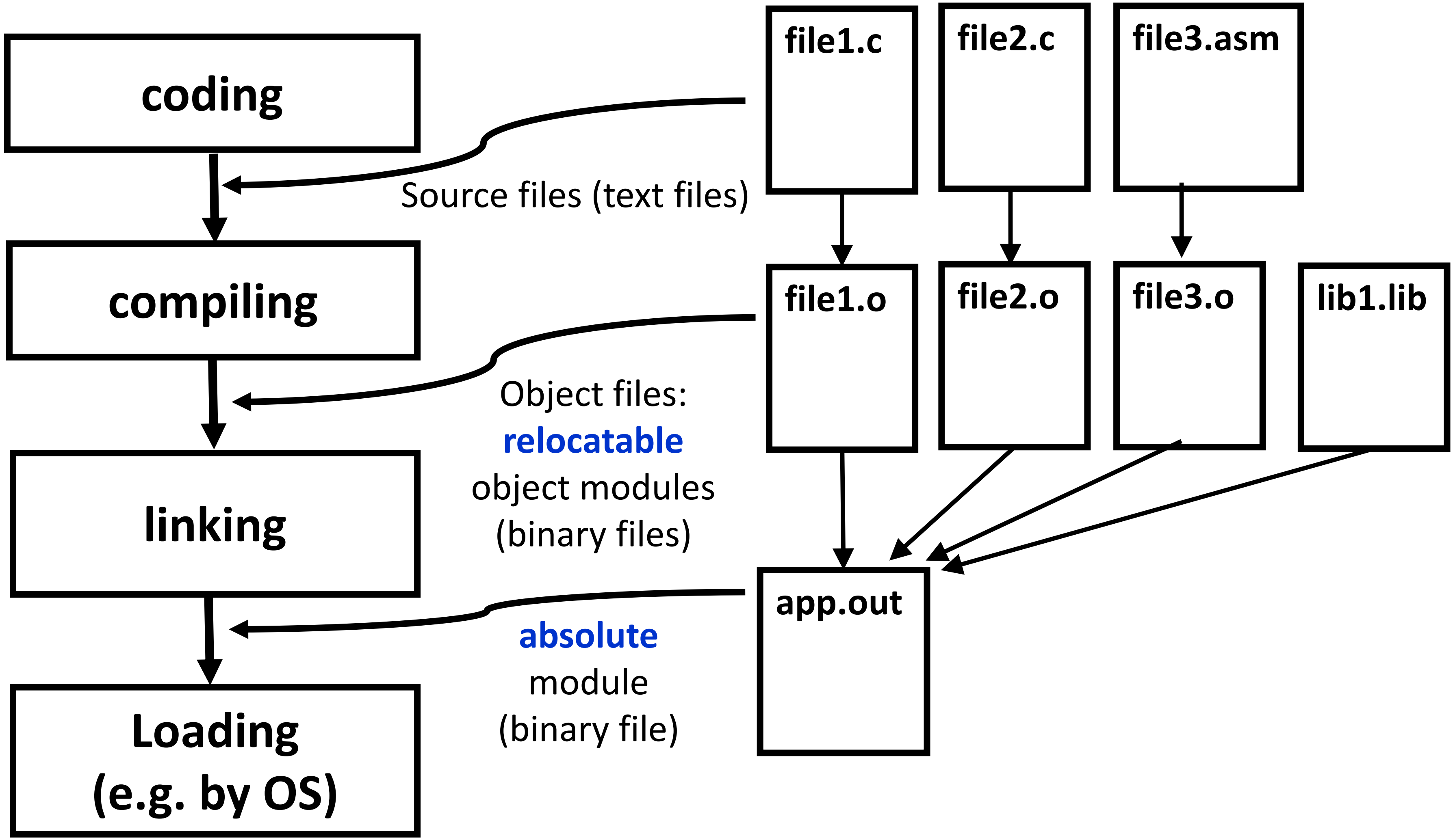
- With condition variables, each procedure ***F*** will then be replaced by

```
wait(mutex);
...
body of F
...
// Before exiting, signal other threads that are
// blocked inside the monitor (i.e blocked
// themselves when using the signal-and-wait for
// the condition variable)
if (next count > 0)
     signal(next);
else
     signal(mutex);
```

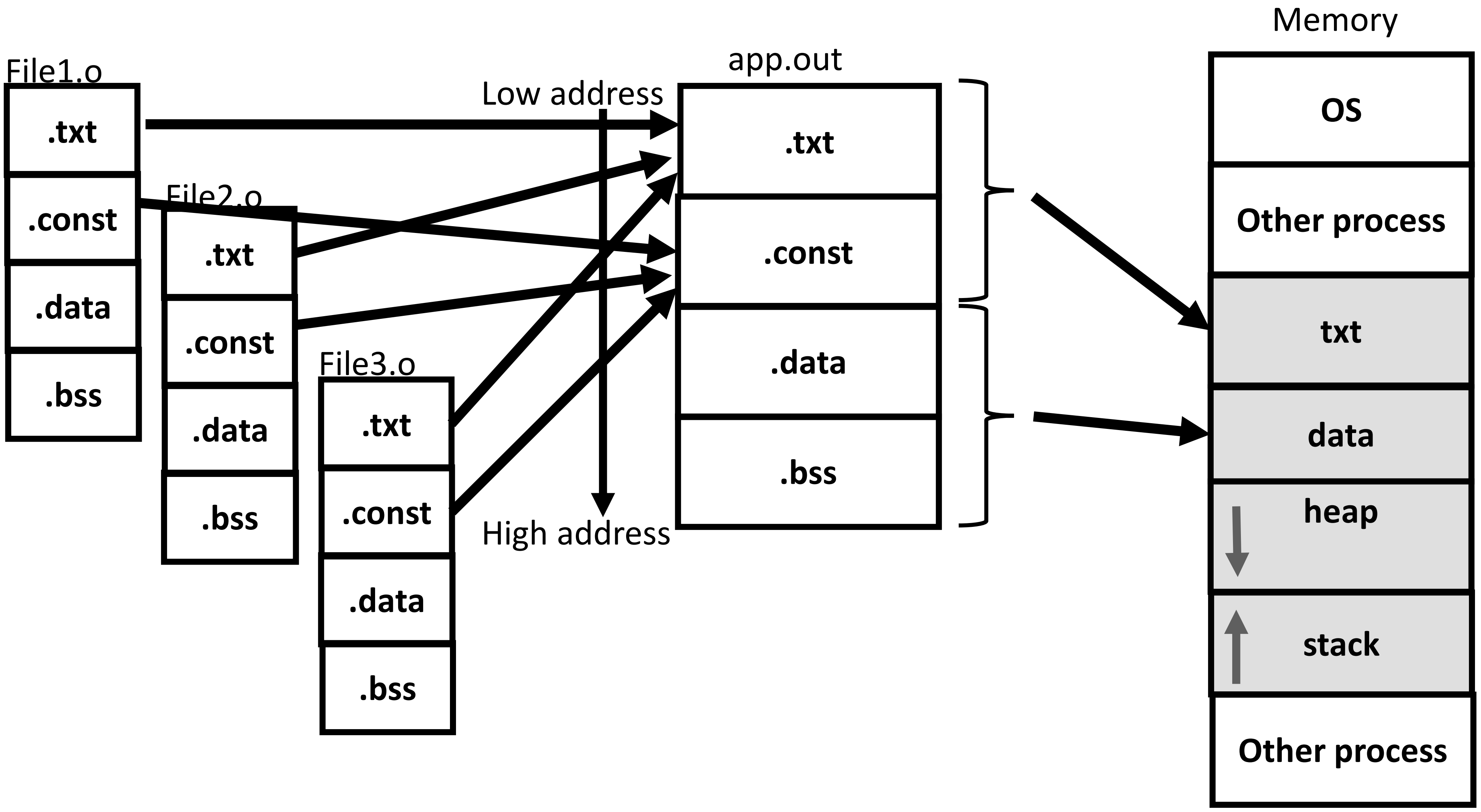
6: Memory Management

- Program translation
- Program relocation
- H/W support for program relocation
- Memory partitioning
- Memory swapping
- Virtual memory
  - Memory segmentation
  - Memory paging
- Virtual Memory management
  - Static virtual memory management and demand paging
  - Dynamic virtual memory management

Program translation



Program translation – linking + loading



Example:

```
static int gVar;

int proc_a(int arg){
        . . .
        gVar = 7;
        put_record(gvar)
}
```

Source file

```
0000        . . .
. . .
0008        [space for gVar]
. . .
0036        Entry      proc_a
. . .
0220        mov R1,#7
0224        st R1, [0008]
0228        push R1
0232        call “put_record”
. . .
0400        External reference table
. . .
0404        “put_record”         0232
. . .
0500        External definition table
. . .
0540        “proc_a”             0036
0600        Symbol table (optional)
. . .
0799        Last location in the module
```

Relocatable object file

```
0000        . . .
. . .
0008        [space for gVar]
. . .
0036        Entry     proc_a
. . .
0220        mov R1,#7
0224        st R1, [0008]
0228        push R1
0232        call "put_record"
. . .
0400        External reference table
. . .
0404        "put_record"        0232
. . .
0500        External definition table
. . .
0540        "proc_a"            0036
0600        Symbol table (optional)
. . .
0799        Last location in the module
```

Relocatable object file

```
0000        Other module
. . .
1008        [space for gVar]
. . .
1036        Entry     proc_a
. . .
1220        mov R1,#7
1224        St R1, [1008]
1228        Push R1
1232        call 2334
. . .
1399        End of proc_a
. . .       Start of Other module
2334        Entry     put_record
. . .
2670        (optional symbol table)
2999        (last location in module)
```

Absolute module

NOTE: Relocatable object files and absolute modules **are binary files** (not text) – Above is only a representation of the actual binary files

Program translation - Relocation issues

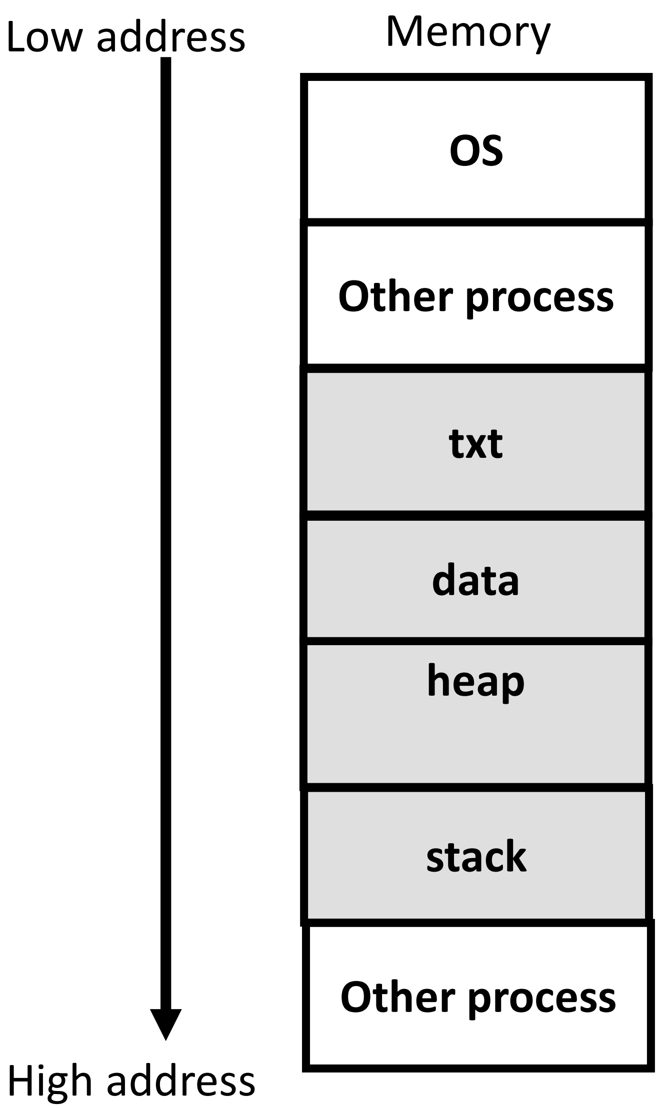
- Referencing static variables:
  - gVar was located at location 0008, but now it is located at 1008. What happens to instructions trying to access gVar ?
  - Clearly, the linker now needs to get to every instruction that accesses gVar and change its reference to location 1008. There are multiple ways of doing this:
    - Linker finds every load or store instruction trying to access the old memory location and changes the address to the new location.
    - Linker uses the optional symbol table, where each relocatable symbol is readily listed and it can easily pin point which instruction is accessing gVar
- Absolute branches
  - ***Relative branches*** are not a problem since they specify an offset from the current program counter location.
  - Linker needs to modify addresses referenced in ***absolute branches*** (or calls) just as it did for static variable, e.g. the call to “put_record” gets replaced with the absolute (aka logical) address of “put_record”.

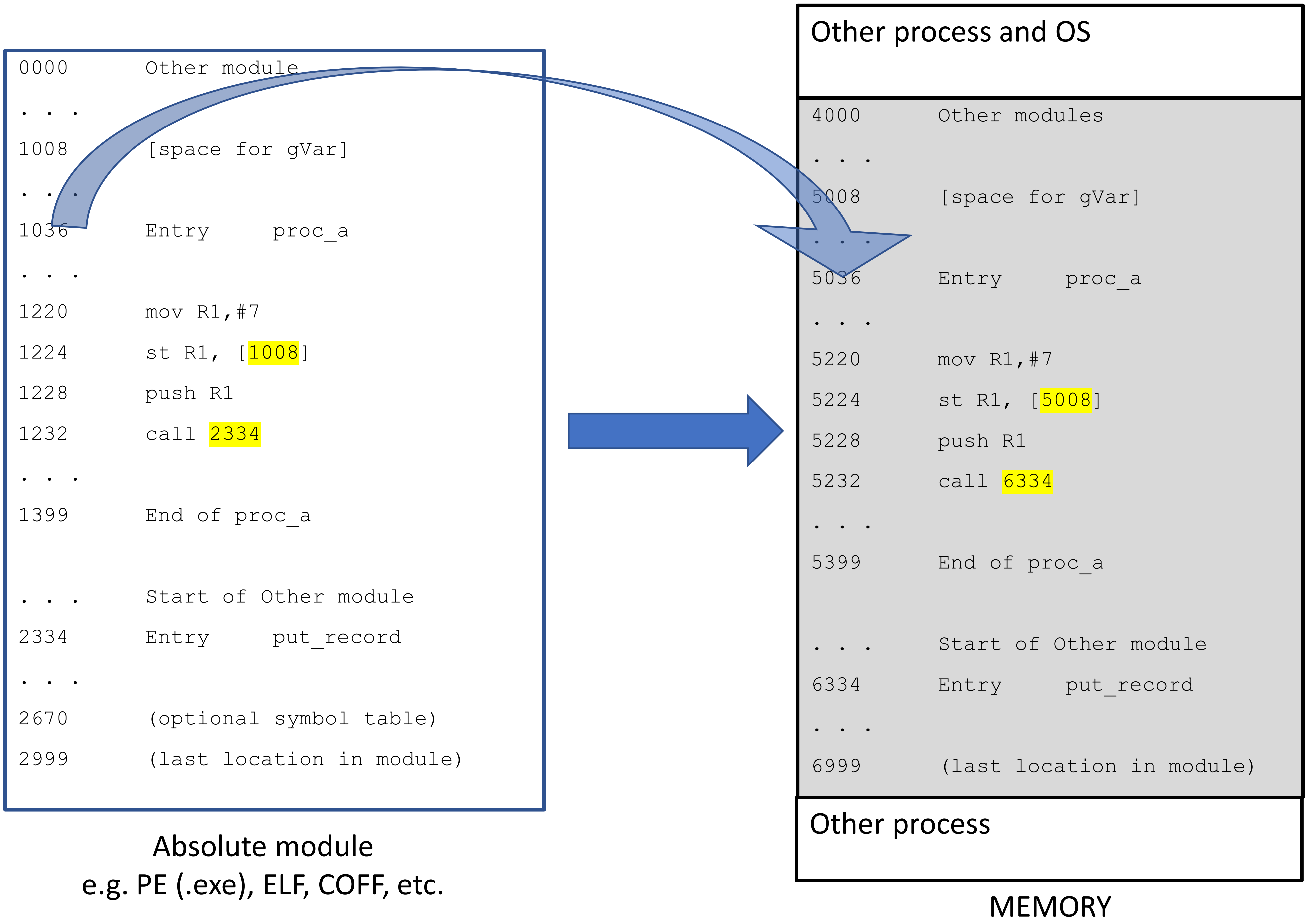
Program translation – file formats

- Source files:
  - Formatted as text files
  - e.g. main.c
- Relocatable object modules and absolute modules:
  - Contain **binary machine instructions** + **information used by linker or loader**
  - Formatted as binary files.
  - Common formats are:
    - COFF: common object file format – commonly used in embedded systems (e.g. microcontrollers: program is loaded into flash memory)
    - PE: portable executable format – used in Windows, extension is .exe
    - ELF: Executable and linkable format – used in Unix/Linux systems.

Program translation – loading

- Memory may be partitioned to accommodate multiple running processes.
- A program is loaded into an area of memory that does not necessarily start at address 0x00.
- Thus more relocation needs to take place at load time.




0000 Other module
. . .
1008 [space for gVar]
. . .
1036 Entry proc_a
. . .
1220 mov R1,#7
1224 st R1, [1008]
1228 push R1
1232 call 2334
. . .
1399 End of proc_a
. . . Start of Other module
2334 Entry put_record
. . .
2670 (optional symbol table)
2999 (last location in module)
Absolute module
e.g. PE (.exe), ELF, COFF, etc.
Other process and OS
4000 Other modules
. . .
5008 [space for gVar]
. . .
5036 Entry proc_a
. . .
5220 mov R1,#7
5224 st R1, [5008]
5228 push R1
5232 call 6334
. . .
5399 End of proc_a
. . . Start of Other module
6334 Entry put_record
. . .
6999 (last location in module)
Other process
MEMORY

Program translation – cont.

- The process of finding an address in the physical memory (physical address) and attaching it to the relocatable address in the absolute module (= logical address as in your text book) is referred to as **binding**. This binding takes place at **load time**.

- Similarly another binding takes place when attaching addresses in the relocatable object module into addresses in the absolute module. This binding takes place at **compile time**.

- Note that in many systems, it may be that the entire system only runs **one program** and the logical address may be the same as the physical address (e.g. a microcontroller), and thus there may be no need for load-time address binding.

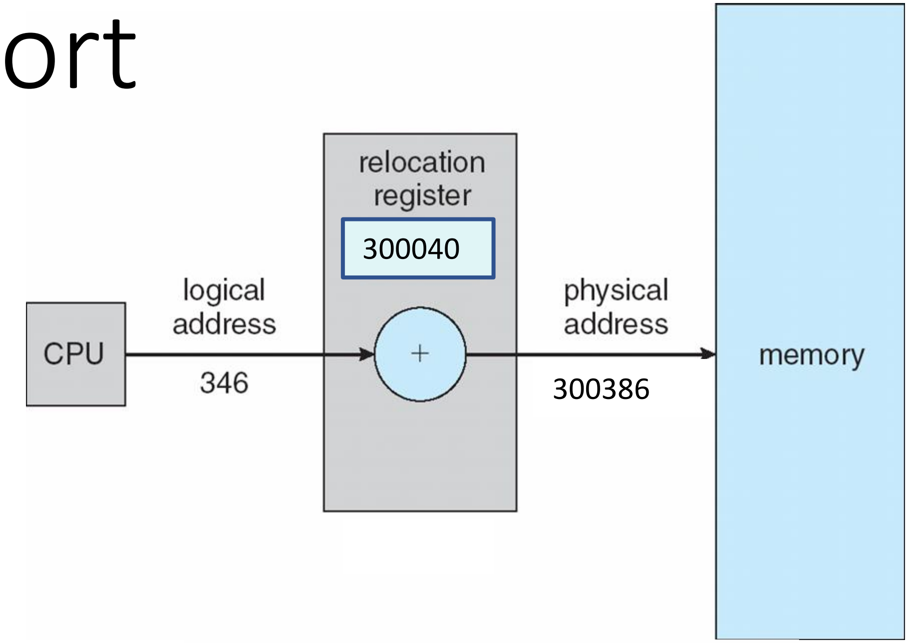
- User programs are designed using the logical address, i.e. the program thinks it is running alone in memory and thinks it starts at address 0.

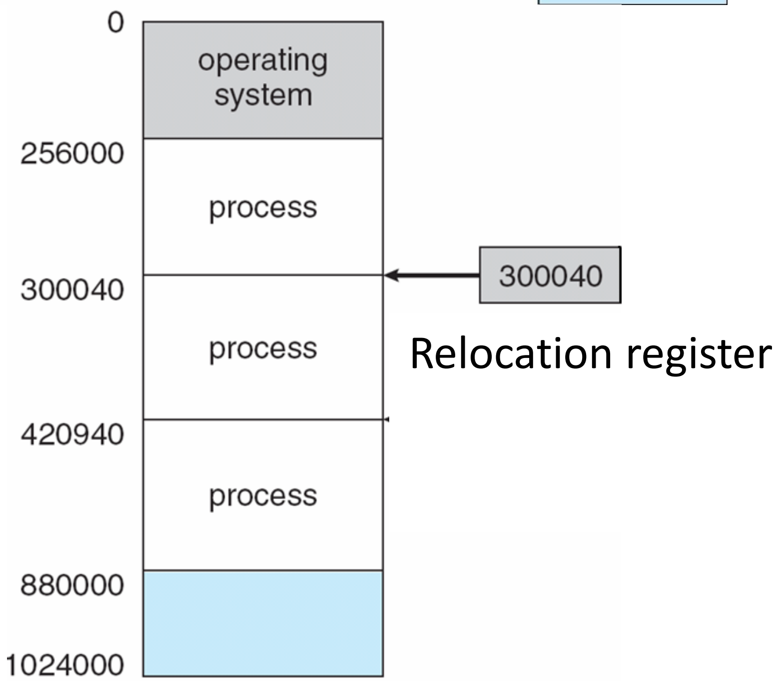
Hardware relocation support

- A **relocation** register may be used to convert the logical address to a physical address. This register may be also referred to as a **base** register.

- Since a program's logical address starts at 0x0000 but needs to be loaded and executed at a different location in the physical memory, a base register may be summed to every logical address before that address is dispatched to the main memory -> thus it does the job of **binding** instead of the loader.

- Many of the older intel CPUs (e.g. 8088) supported 4 segment registers for: code, data, stack and extra segments.





Hardware Memory protection

- CPU may check every physical memory access generated by a process to be sure it is between **base** and **limit** registers. This hardware thus performs **memory protection**.

- If a process attempts to access an address that is outside its limits, an **exception** takes place and the OS kernel is invoked.

- OS is responsible for setting the correct values in base and limit registers for each process prior to loading it. OS is also in charge of saving that information in the **PCB** during **context switching**.