Synchronization Techniques (2)

More on Synchronization

Hardware Support for Synchronization

Very much needed for multi-core processors

Memory barriers

- A system (processor or compiler) can reorder instructions for performance
- Break software solutions to synchronization
- Computer architectures determines what guarantees memory can provide the applications - **memory model**

Hardware instructions

- Instructions in the processor to test and set memory locations **atomically** - one uninterruptible unit
- Micro processors provide a variety of these instructions - accessible in assembly programming

Atomic variables

- This is the way hardware instructions are exposed to the programmer in a programming language
- Use this to build higher level sync primitives
- Atomic variables themselves may not solve all sync issues

Memory Barriers

Also referred to as memory fences

- Relate to shared memory - determines how the loads and stores to the shared memory is visible to the processors that are doing them
- Two type of memory barriers
  - **Strongly ordered** - memory modification on one processor is immediately visible to all other processors
  - **Weakly ordered** - memory modifications to memory in one processor is not immediately visible to another processor

```
boolean flag = false;
int x = 0;
```

where Thread 1 performs the statements

```
while (!flag)
   ;
print x;
```

and Thread 2 performs

```
x = 100;
flag = true;
```

```
while (!flag)
  memory_barrier();
print x;
```

```
x = 100;
memory_barrier();
flag = true;
```

Hardware Instructions

- **Test_And_Set (TAS)** - retrieve the current value of a memory and set it to 1 in an uninterruptible instruction
- Implemented by the processor
- Executes a **read-modify-write** atomically

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

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

      /* critical section */

   lock = false;

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

Hardware Instructions

- Compare_And_Swap (CAS) is like TAS, but uses a different mechanism
- It operates on three operands: it sets the memory variable to the new value only if the current value is same as expected - whole operation is atomic - all or none
- Two CAS instructions executed concurrently, they are ordered for execution sequentially in an arbitrary manner

```
int compare_and_swap(int *value, int expected, int new_value) {
  int temp = *value;

  if (*value == expected)
    *value = new_value;

  return temp;
}
```

```
while (true) {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ; /* do nothing */

    /* critical section */

  lock = 0;

    /* remainder section */
}
```

Examples using Hardware Instructions

- Implement a mutual exclusion using CAS
- Critical section is protected by a lock - a memory variable
- Threads wanting to enter the CS need to grab the lock
- Only one winning thread - all other threads are waiting to get the lock using busy waiting

```
while (true) {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ; /* do nothing */

    /* critical section */

  lock = 0;

    /* remainder section */
}
```

Atomic Variables

- Hardware instructions are not used directly
- They are exposed via atomic variables that operate on integers, booleans, etc
- Lets say we want to implement
- increment() on an atomic variable
- atomic variables solve part of the synchronization issues in an application such as producer-consumer

```
void increment(atomic_int *v)
{
   int temp;

   do {
      temp = *v;
   }
   while (temp != compare_and_swap(v, temp, temp+1));
}
```

Mutex Locks

- OS designers provide software tools to solve CS
- Simplest is mutex locks
- We enter the CS by acquiring the lock
- We release the lock at exit
- Calls to acquire() and release() must be atomic
- If the solution depends on busy waiting, locks are called spin locks

```
while (true) {

    acquire lock

        critical section

    release lock

        remainder section

}
```

The definition of `acquire()` is as follows:

```
acquire() {
   while (!available)
     ; /* busy wait */
   available = false;
}
```

The definition of `release()` is as follows:

```
release() {
   available = true;
}
```

Lock Contention & Duration

Lock contention

- Lot of threads can try to grab a lock at the same time
- All except one thread would win and others have to wait
- Overhead at lock acquiring is known as the lock contention - time to grab the lock
- Lock contention rises with increasing number of waiting threads - all wanting to get into the CS at the same time

Lock duration

- Duration for which the lock is going to be held by the acquiring thread
- Short duration locks can use busy waiting with good overall performance

Semaphore

A generalized synchronization primitive

- Mutex locks are used to protect access to the critical sections
- Semaphores provide generalized synchronization for processes (threads) - that can be used to implement mutex locks
- Semaphore was invented by Edsger Dijkstra
- It is a special variable (s) supported by the OS - implemented inside the kernel
- Supports two operations - wait(s) (P(s)) and signal (s) (V(s))
- Two or more processes are cooperate by manipulating the semaphores
- wait() and signal() is considered as message passing using the semaphore variable
- Complex coordination mechanisms can be implemented

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

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

Semaphore Types

Counting semaphore

- Initial value set to any non negative integer value
- Internal value can be negative

Binary semaphore

- Initial value can be set to either 0 or 1
- Internal value can be negative or restricted to 0 (min) - book vague on this implementation issue
- Strong semaphore - if a strict FIFO order is enforced in serving the blocked processes
- Weak semaphore - no strict FIFO order in serving blocked processes

```
struct semaphore {
        int count;
        queueType queue;
}
```

```
void wait(semaphore s)
{
    s.count--;
    if (s.count < 0)
    {
        place this process in
        s.queue;
        block this  process
    }
}
```

```
void signal(semaphore s)
{
    s.count++;
    if (s.count <= 0)
    {
        remove a process P
        from s.queue;
        place process P on
        ready list;
    }
}
```

Producer Consumer Using Semaphores

- Implement producer-consumer using 3 semaphores
- Protect the critical section that does queue manipulation
- Buffer overflow synchronization
- Buffer underflow synchronization

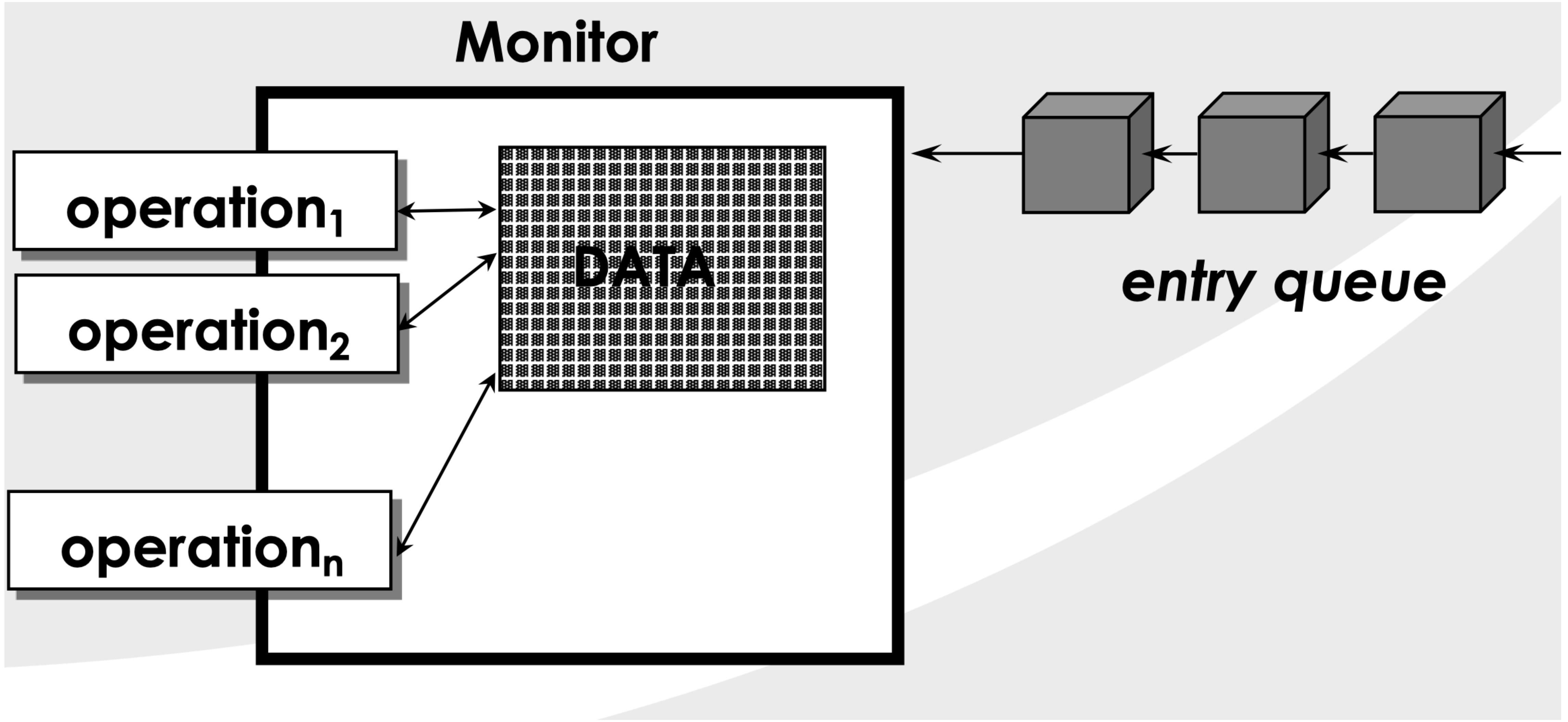
```
semaphore mutex = 1;      // protects the critical section
semaphore empty = N;      // counts the empty slots
semaphore full = 0;       // counts full buffer slots


producer() {                          consumer() {
    int item;                             int item;

    while(TRUE) {                         while(TRUE) {
        item = produce_item();                wait(&full);
        wait(&empty);                         wait(&mutex);
        wait(&mutex);                         item = remove_item();
        insert_item(item);                    signal(&mutex);
        signal(&mutex);                       signal(&empty);
        signal(&full);                        consume_item(item);
    }                                     }
}                                     }
```

Monitors

A higher level abstract data type that provides synchronization

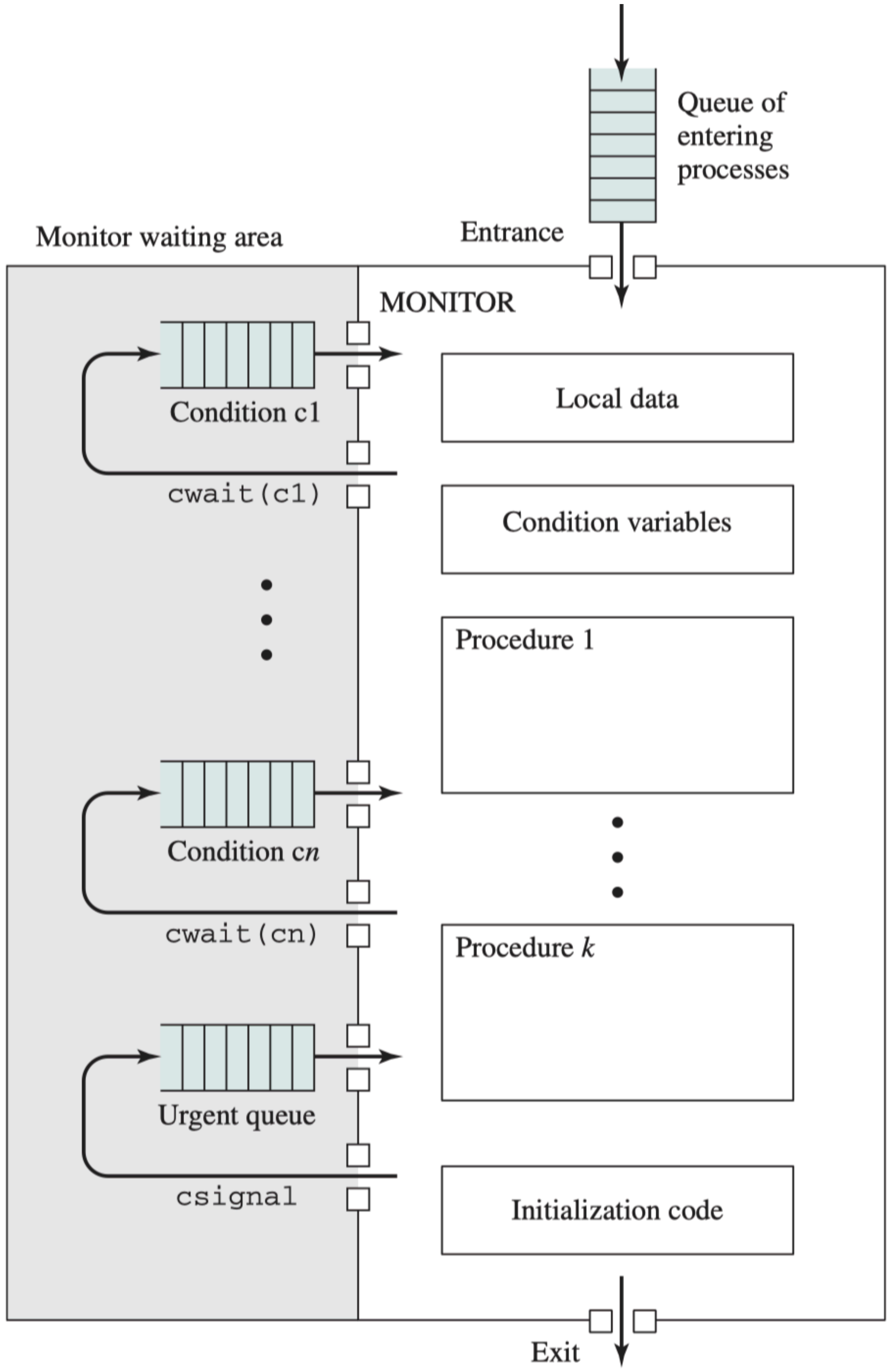
- A higher level abstraction that provides a convenient and effective mechanism for synchronization
- Abstract data type that encapsulates synchronization variables so they are manipulatable by code within the ADT

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

  function P2 (…) { …. }

  function Pn (…) {……}

  initialization code (…) { … }
}
```



Monitor in More Detail

- Monitor provides a critical section - only one thread can be active inside it
- Monitor provides facilities for threads to sleep inside the monitor - such threads are not active inside the monitor
- You enter the monitor by calling a method exposed by the monitor
- A thread signalling inside the monitor goes to sleep inside the monitor



Monitor Example

- Producer-consumer using monitor
- Has two condition variables
  - Full buffer — notfull
  - Empty buffer — notempty
- No need to protect queue insertion or deletion!

Monitor Definition

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                        /* space for N items */
int nextin, nextout;                                     /* buffer pointers */
int count;                                        /* number of items in buffer */
cond notfull, notempty;                /* condition variables for synchronization */

void append (char x)
{
      if (count == N) cwait(notfull);           /* buffer is full; avoid overflow */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;
      /* one more item in buffer */
      csignal (nonempty);                          /*resume any waiting consumer */
}
void take (char x)
{
      if (count == 0) cwait(notempty);        /* buffer is empty; avoid underflow */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                       /* one fewer item in buffer */
      csignal (notfull);                          /* resume any waiting producer */
}
{                                                             /* monitor body */
      nextin = 0; nextout = 0; count = 0;                /* buffer initially empty */
}
```

Using the monitor

```
void producer()
{
      char x;
      while (true) {
      produce(x);
      append(x);
      }
}
void consumer()
{
      char x;
      while (true) {
      take(x);
      consume(x);
      }
}
void main()
{
      parbegin (producer, consumer);
}
```