1. What parts of the L.M.C. correspond to the elements of the Von Neumann architecture

Von Neumann defined the functional organization of the computer to be made up of:

* + A *control unit* that executes instructions
  + An *arithmetic logic unit* that performs arithmetic and logical calculations,
  + Memory locations
  + Input/output
  + Communication bus

1. Describe how the LMC performs the Fetch and execute cycle

Fetch: Little Man finds out what instruction he is to execute

1. Little Man reads the address from the location counter

2. He walks over to the mailbox that corresponds to the location counter

3. And reads (involves knowing the meaning of the op code…) the contents on the slip of paper (he puts the slip back in case he needs to read it again later)

Execute: Little Man performs the work he is instructed to perform

1. The Little Man goes to the mailbox address specified in the instruction he just fetched.
2. He reads the number in that mailbox (he remembers to replace it in case he needs it later).
3. He walks over to the calculator and punches the number in.
4. He walks over to the location counter and clicks it, which gets him ready to fetch the next instruction.
5. Wait()

Processes may be in one of 5 states, as shown in Figure 3.2 below.

New - The process is in the stage of being created.

Ready - The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.

Running - The CPU is working on this process's instructions.

Waiting - The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. For example the process may be waiting for keyboard input, disk access request, inter-process messages, a timer to go off, or a child process to finish.

Terminated - The process has completed.

1. What is a node?

A node had two parts:

one for containing data (an integer; an employee record....)

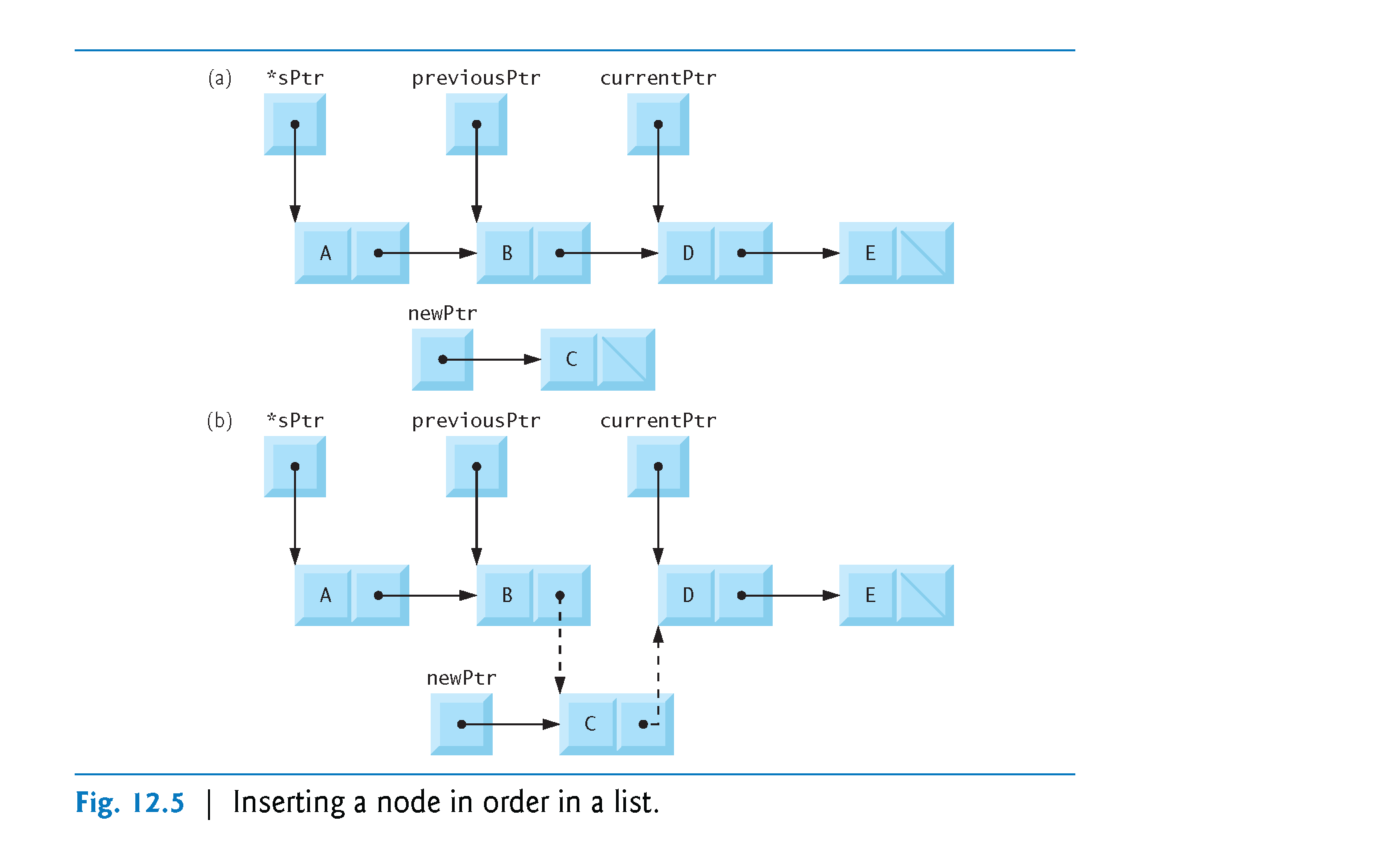
A pointer to another node (link between nodes)

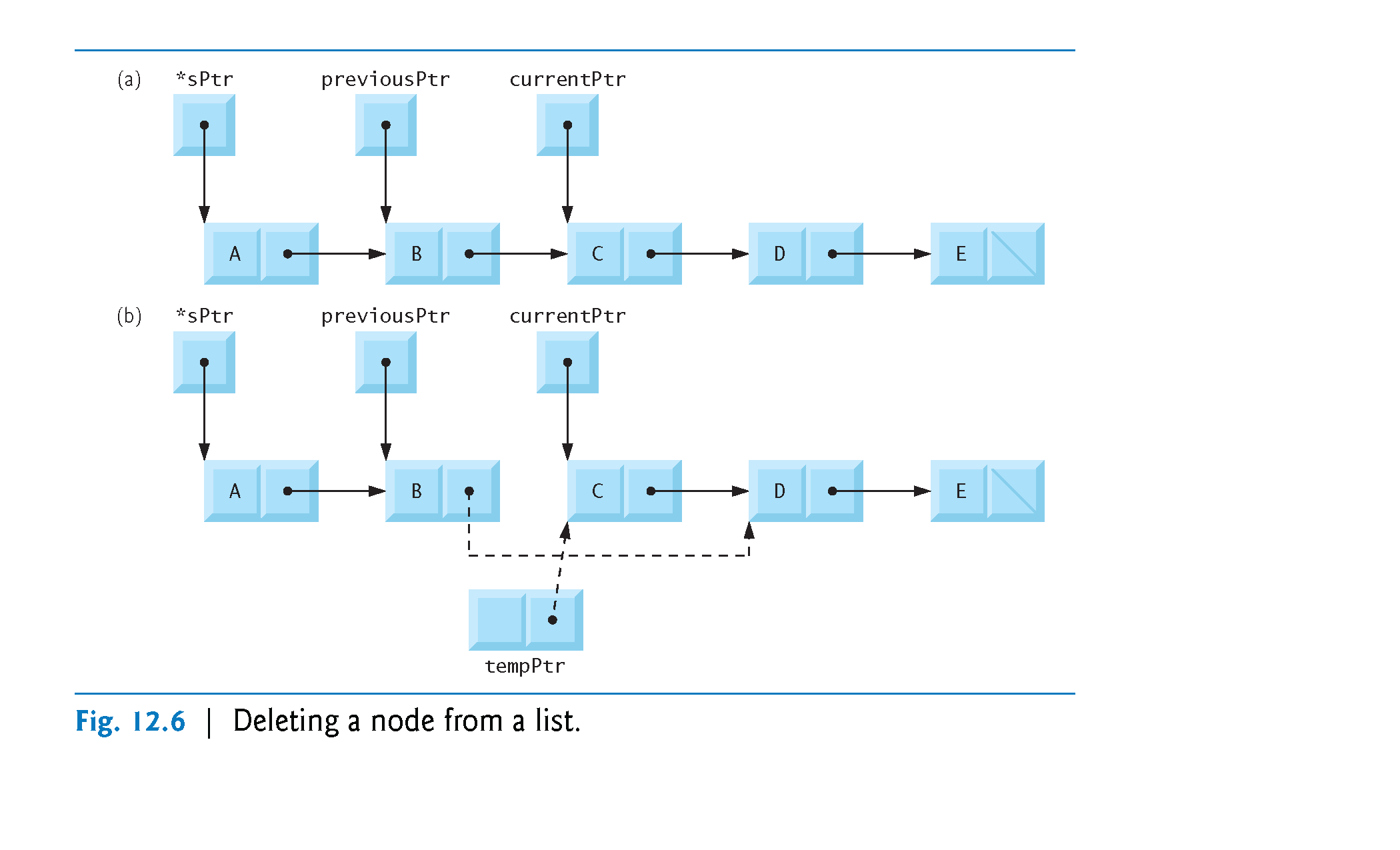
1. LINK LIST

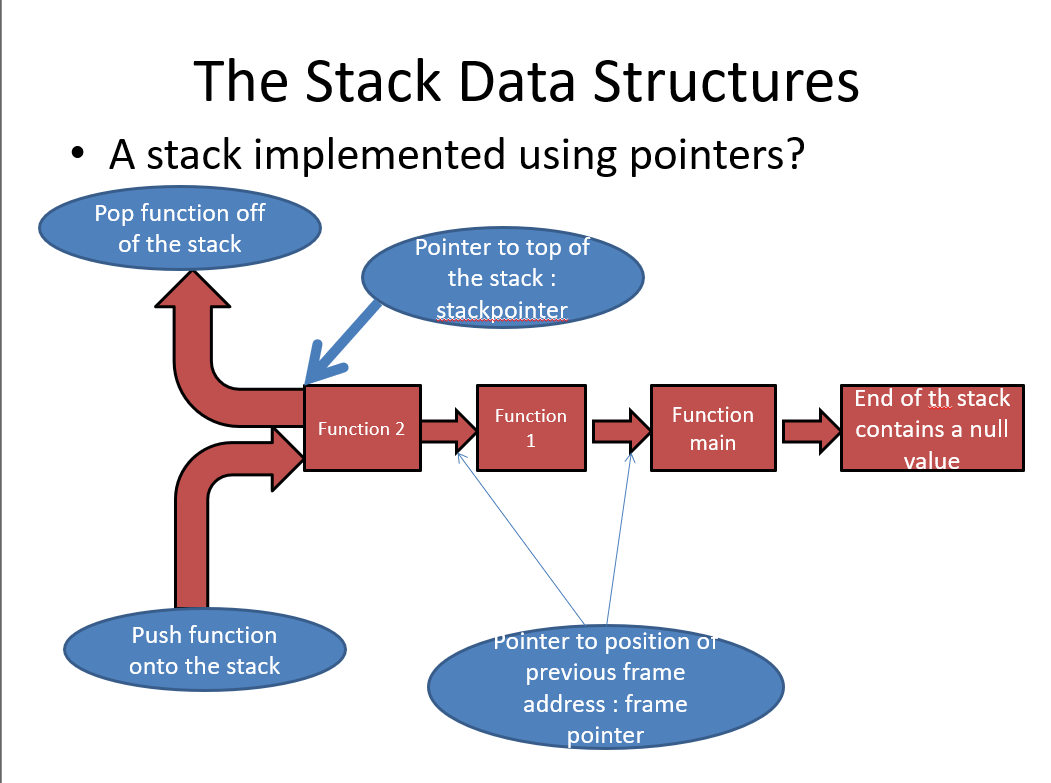
Each node contains at least

A piece of data (any type; e.g. a structure – P.C.B.)

Pointer to the next node in the list

It has an address (&node)



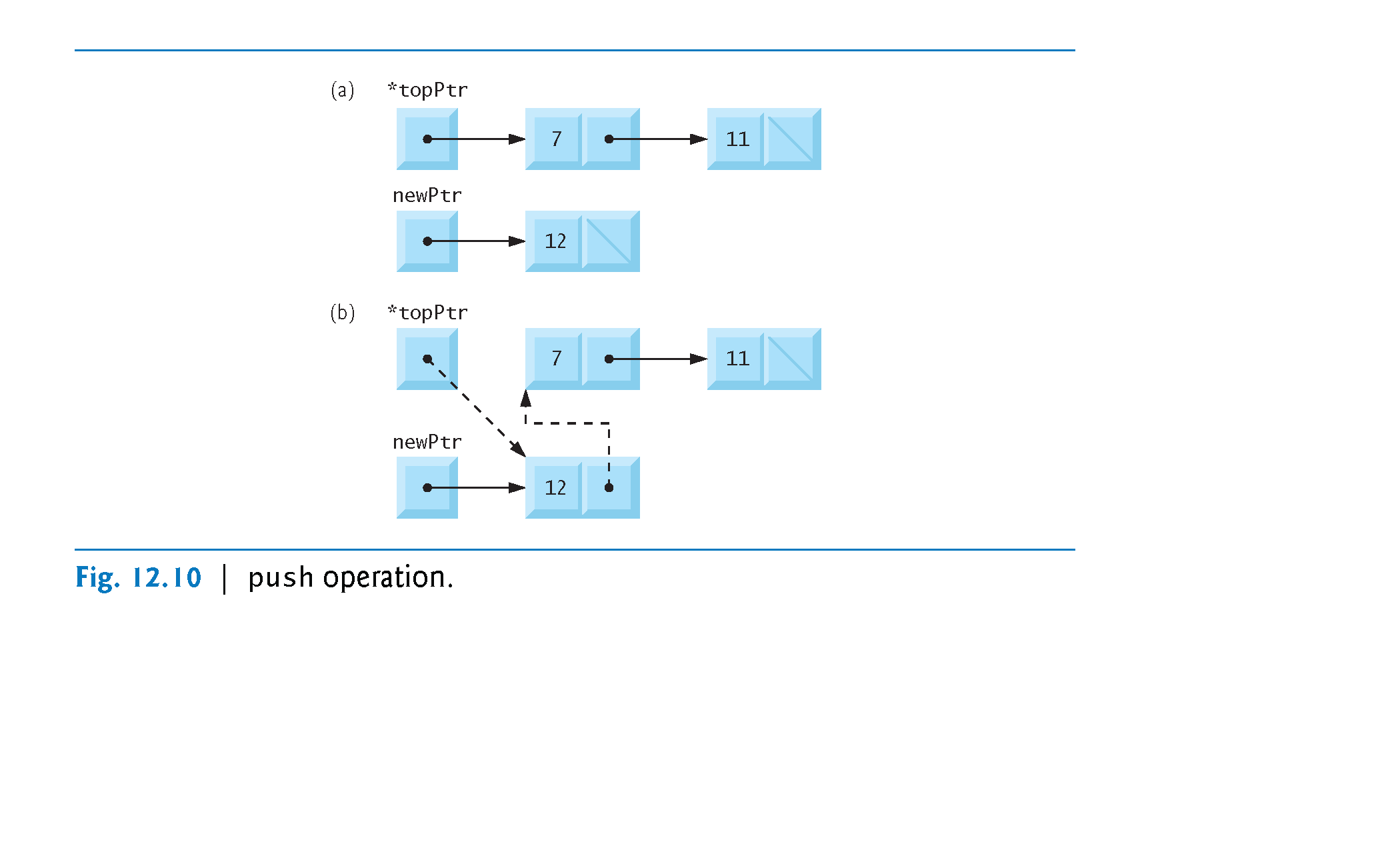
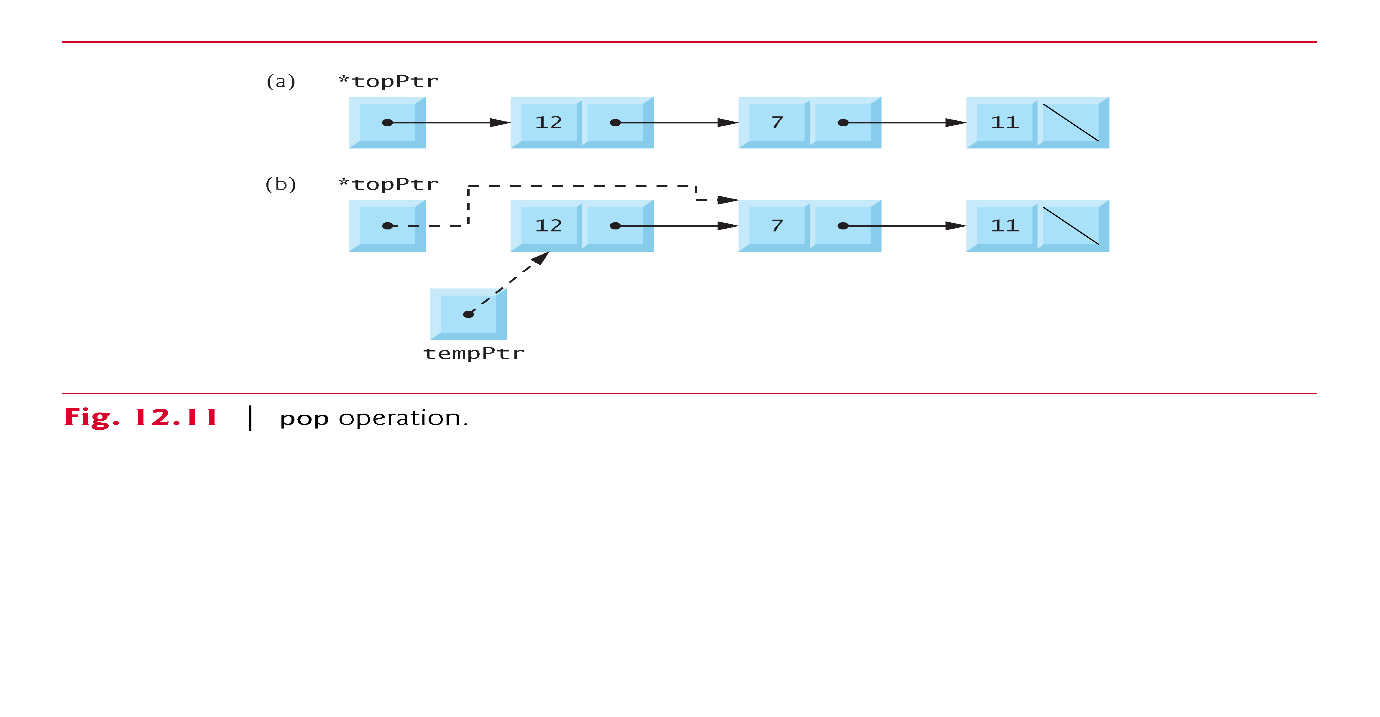
1. STACK

* Two basic functions associated with stacks:
  1. The **push** function: this adds a “node” to the beginning of a stack

The **pop** function: removes a “node” from the beginning of a non-empty stack

1. PUSH STACK

The function consists of three steps:

1. Create a new node and assign the location of the allocated memory to newPtr
2. Assign data, that is to be added to the stack, to the new node; Assign the pointer part of the new node to point to the top of the stack. (newPtr ->nextPtr = …..
3. Move TopPtr to point to new node. {assign address of new node to the TopPtr}
4. POP STACK
5. Assign the data element of node X to a variable data
6. Assign head pointer to point to second node in the stack by Assign it the value stored in the pointer element of the first node.
7. Return data and delete node X
8. Threads

Single threading

* 1. This is a process with only one thread of control so is essentially a single process or one processing one command at a time.

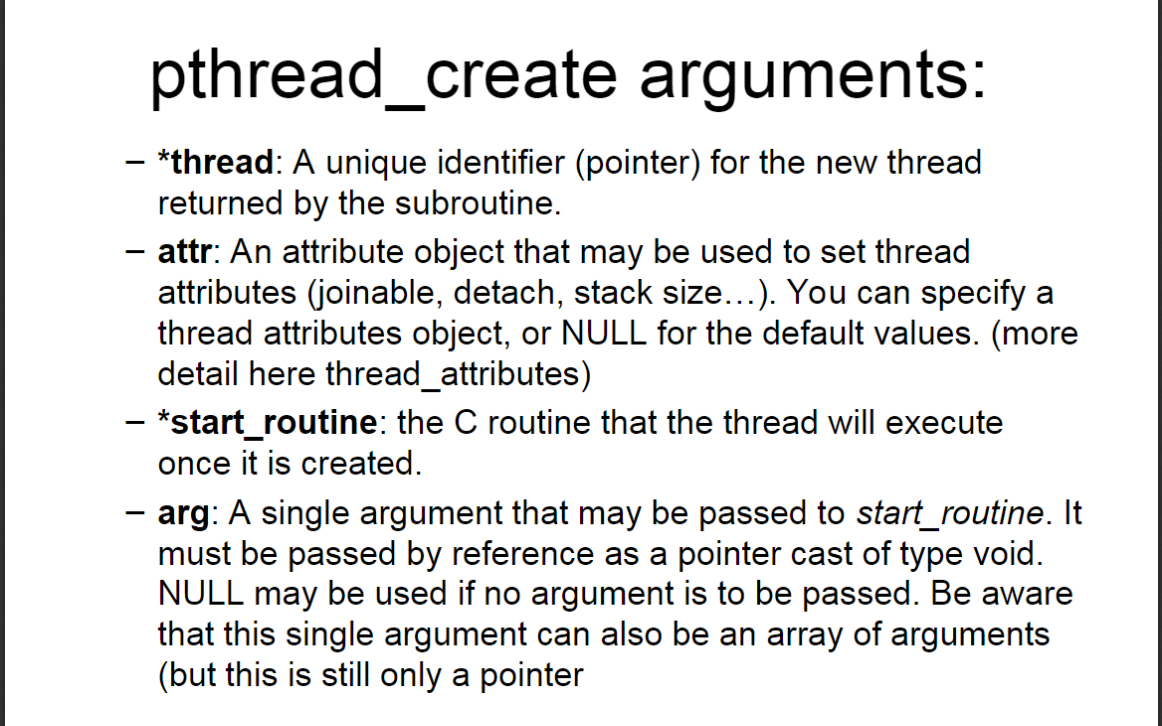
Multithreading

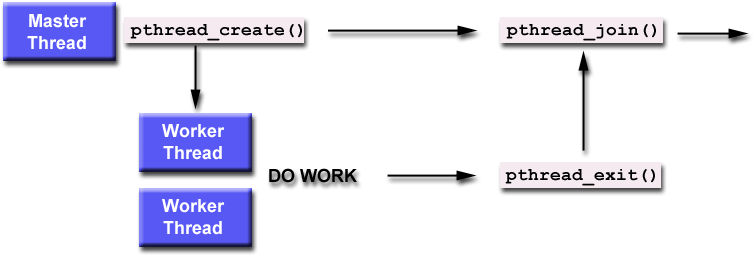
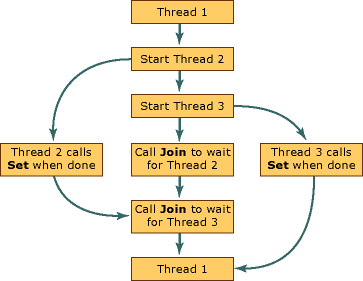
* 1. Allows applications to manage a, separate, process with several threads of control; This allows the OS to emulate a system with multiple processors and a common memory.
  2. In reduces overheads associated with swapping the whole process in and out of memory.
  3. *In other words different parts of the program can be run concurrently in a single processor system or in parallel in a multi-processor system.*

1. Thread states

Operating systems must be able to:

* 1. **Create** new threads,
  2. Set up a thread so it is ready to execute
  3. **Delay**, or put to sleep, threads for a specified amount of time
  4. **Block**, or suspend, threads waiting for I/O to be completed
  5. Set threads to a **WAIT** state until a specific event occurs: until another thread finishes execution
  6. **Schedule** threads for execution
  7. **Terminate** a thread and release its resources





1. Thread Joining#
   1. Joining is similar to wait()
   2. "Joining" is one way to accomplish synchronization between threads.
   3. The pthread\_join() subroutine blocks the calling thread until the specified threadid thread terminates.
   4. On success, pthread\_join() returns 0; on error, it returns an error number.
   5. The thread specified by thread must be joinable.
   6. This is a way to ensure that the process itself does not terminate before the threads and so destroy (not execute) the threads
2. RACE PROBLEM

if two processes or threads/transactions try to update the same object can result in: Race/lost update problem

Race (lost update problem) between processes

–Results when locking not used and no concurrency enforcement policy

–Causes incorrect final version of data

–Depends on process execution order

–Prevented by using concurrency locking (refer to lecture mutexs: test-set, wait and signal and semaphores

1. Test and Set

T.S Executed in single machine cycle

–Test If key available: set to unavailable

Actual key often referred as the Mutex

–Single bit in storage location: zero (free) or one (busy)

Before process enters critical region

–Tests condition code using TS instruction

–If No other process in region (key is zero)

Process proceeds

Condition code (key) changed from zero to one

P1 exits: code (key) reset to zero, allowing others to enter

Use a lock variable (mutex):

while (test\_and\_set(mutex) == 1) {

// do nothing

}

critical\_section();

mutex = 0;

Requires an atomic test-and-set operation

–If mutex value is 0 resets mutex to value 1

–Does not enter the loop; goes to critical section()

–Otherwise stays within the while loop resulting in what is referred to as busy waiting

Advantages

–Simple procedure to implement

–Works well for small number of processes

Drawbacks

–Busy waiting

•Waiting processes remain in unproductive, resource-consuming wait loops

–Starvation (will cover in more detail in next lecture)

•Many processes waiting to enter a critical region

•Processes gain access in arbitrary fashion

