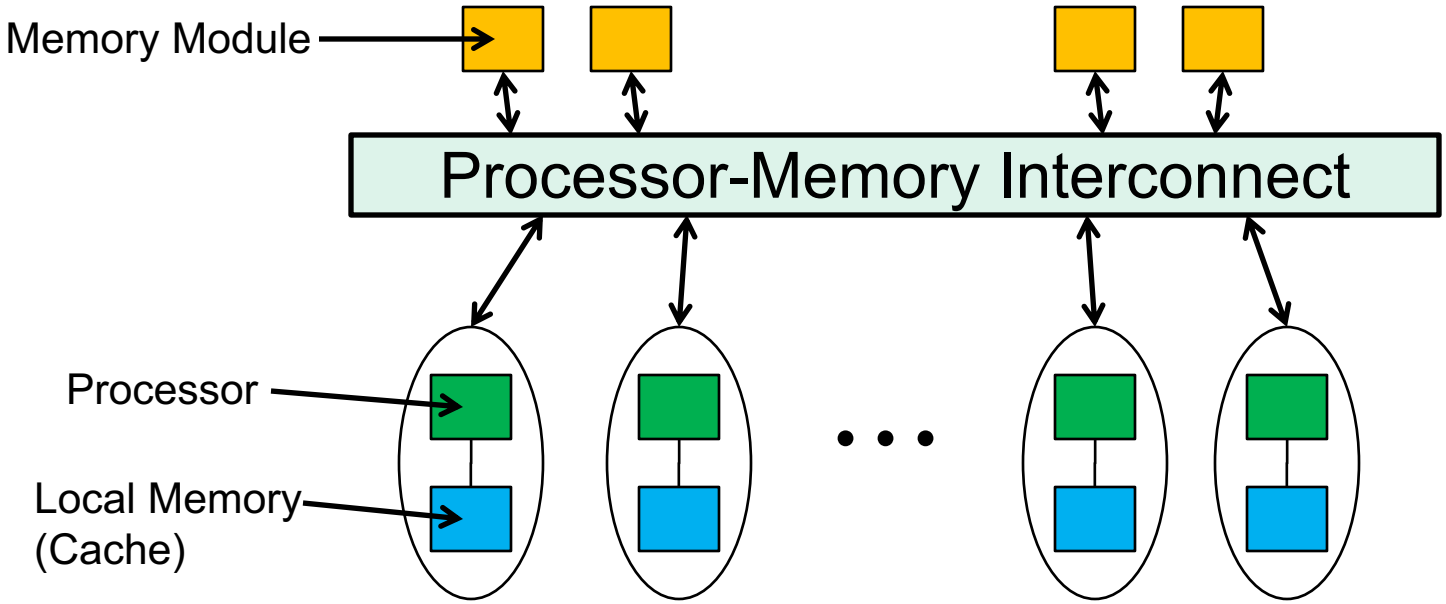
Parallel Computing Using Shared Memory

CPSC 424/524
Lecture #5
September 19, 2018

Types of Parallel Computers

- Shared Memory Multiprocessor
  - Many processors accessing a shared memory



Examples: Multiple independent processors (UMA, NUMA)
Multi-core computers (with multi-level caches)

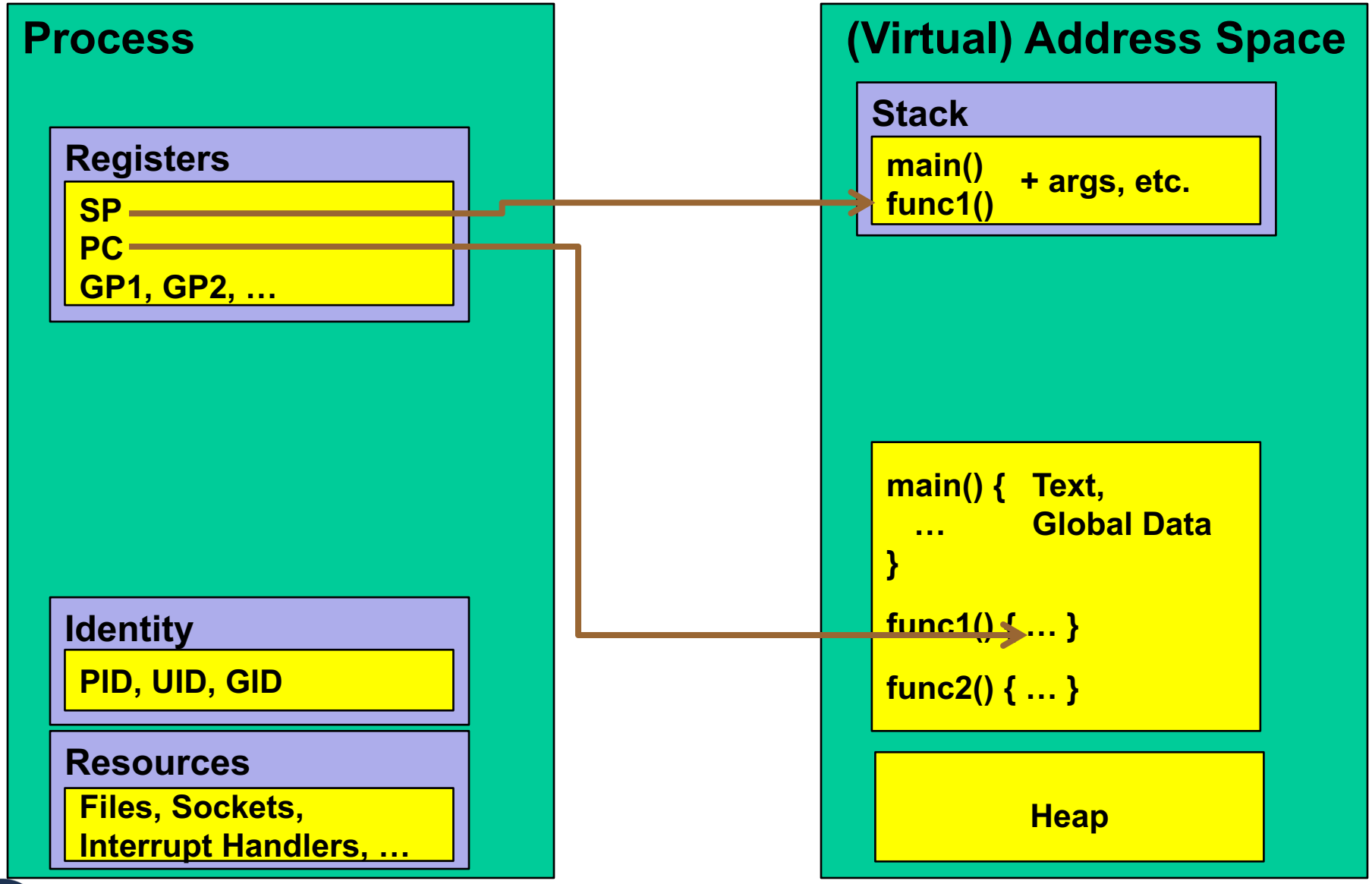
Characteristics of SMPs

- Any memory location can be accessed by any of the physical processors/cores (though not necessarily at a constant cost).
- A single address space exists. Each memory location has a unique address within a single range of addresses.
- By default, data in memory on a shared memory multiprocessor system are available to all processes.
  - Major concern is managing possibly conflicting data accesses
  - Correct programming may require "privatizing" some data to prevent access by others.
- SMP programming often seems easier and more convenient than other approaches, but it requires careful control

Processes and Threads

- Operating systems are based on a notion of a process that fully encapsulates a running (or suspended) computation.
  - Everything private: call stack, stack pointer (SP), registers, program code, program counter (PC), memory heap, files, interrupt handlers, …

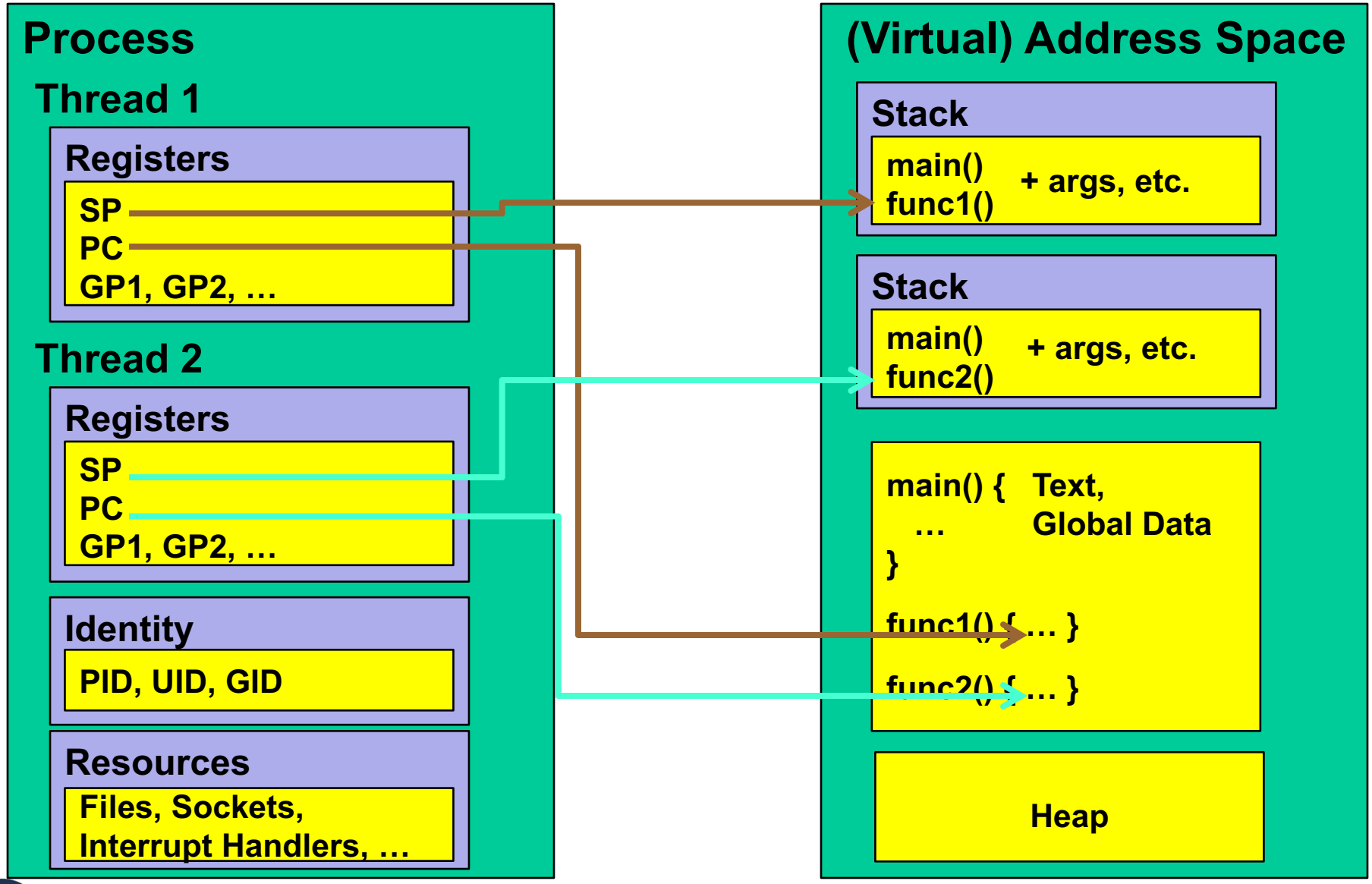
Processes


Process
Registers
SP
PC
GP1, GP2, …
Identity
PID, UID, GID
Resources
Files, Sockets,
Interrupt Handlers, …
(Virtual) Address Space
Stack
main()
func1()
+ args, etc.
main() {
…
}
func1() { … }
func2() { … }
Text,
Global Data
Heap

Processes and Threads

- Operating systems based upon notion of a process that fully encapsulates a running (or suspended) computation.
  - Everything private: call stack, stack pointer (SP), registers, program code, program counter (PC), memory heap, files, interrupt handlers, …
- OS "time shares" the core(s) among multiple processes, switching among them as required
  - Possibly at regular intervals or when active process must wait
  - Process switching requires saving the process state: SP, registers, PC, interrupt status, I/O status, etc. (may be very time consuming)
- A thread is analogous to a "lightweight" process
  - Multiple threads may be children of a full process
  - Sibling threads share the heap, the program code, files, interrupt handlers, sockets, etc.

Processes and Threads

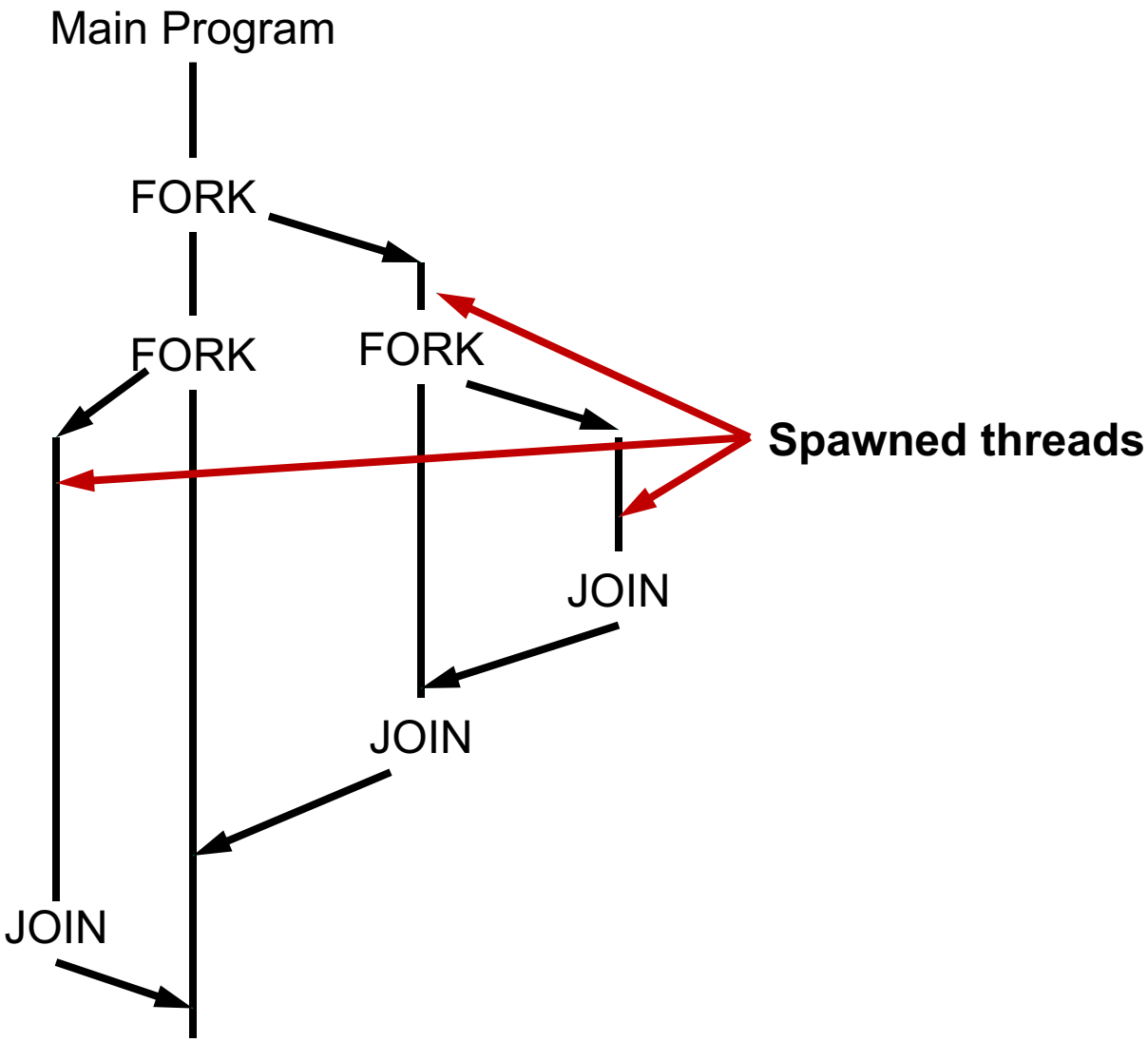


Programming Shared Memory Multiprocessors

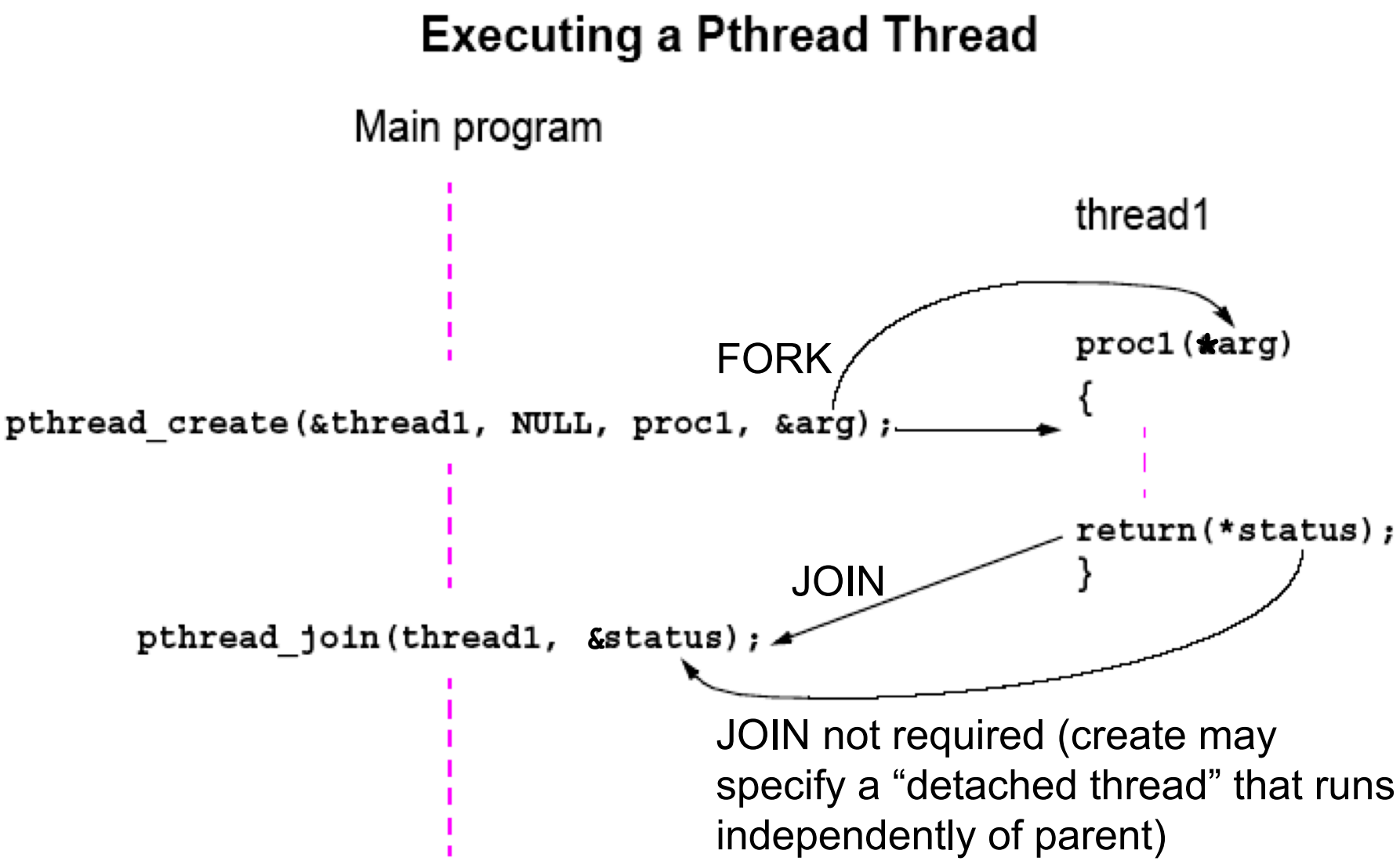
1. Multiple full processes
2. Multiple threads handled "natively" in a sequential language (e.g. Pthreads in C/C++)
3. Multiple threads handled using "preprocessor annotations" ("pragmas") in a sequential language (e.g. OpenMP)
4. Specially modified sequential language (e.g. HPF)
5. Special parallel languages
6. Parallelizing compilers

Approaches 2-6 typically use multiple threads, but provide different tradeoffs of performance vs. ease of use.

Fork-Join Model for Thread-Based Parallelism



Pthreads: IEEE POSIX Standard



Statement Execution Order

- Single core: Processes/threads typically executed in order until blocked.
- Multicore: Instructions of processes/threads may be interleaved in time.
- Compilers and the hardware may cause unexpected execution ordering

Example

Process 1	*Process 2*
Instruction 1.1	Instruction 2.1
Instruction 1.2	Instruction 2.2
Instruction 1.3	Instruction 2.3

Many possible orderings, e.g.:

Instruction 1.1
Instruction 1.2
Instruction 2.1
Instruction 1.3
Instruction 2.2
Instruction 2.3

assuming instructions cannot be divided into smaller parts.

Anomalies with Shared Data

Instruction	Thread 1	Thread 2	Thread 3
	`k = 1;`	`k = 2;`	`k = 3;`
`x = x + k;`	`load x`	`load x`	`load x`
	`compute x=x+k`	`compute x=x+k`	`compute x=x+k`
	`store x`	`store x`	`store x`

Suppose that **`x`** is shared and **`k`** is not, and that **`x`** starts with value 1. What is the value of **`x`** after the "pseudo assembler" code executes?

To avoid problems, only 1 thread at a time should update shared data.

Thread-Safe Routines

- Routines are "thread safe" if they can be called from multiple threads simultaneously while always producing correct results
  - Example: Standard I/O operations are thread-safe (e.g., multiple threads can print messages without printing interleaved characters)
  - Not all system routines are necessarily thread-safe
- Routines using shared data require special care to be thread safe
  - Is this code (similar to code on the previous slide) thread safe?

```
k = 1;
read x;
x = x + k;
write x;
```

Critical Sections

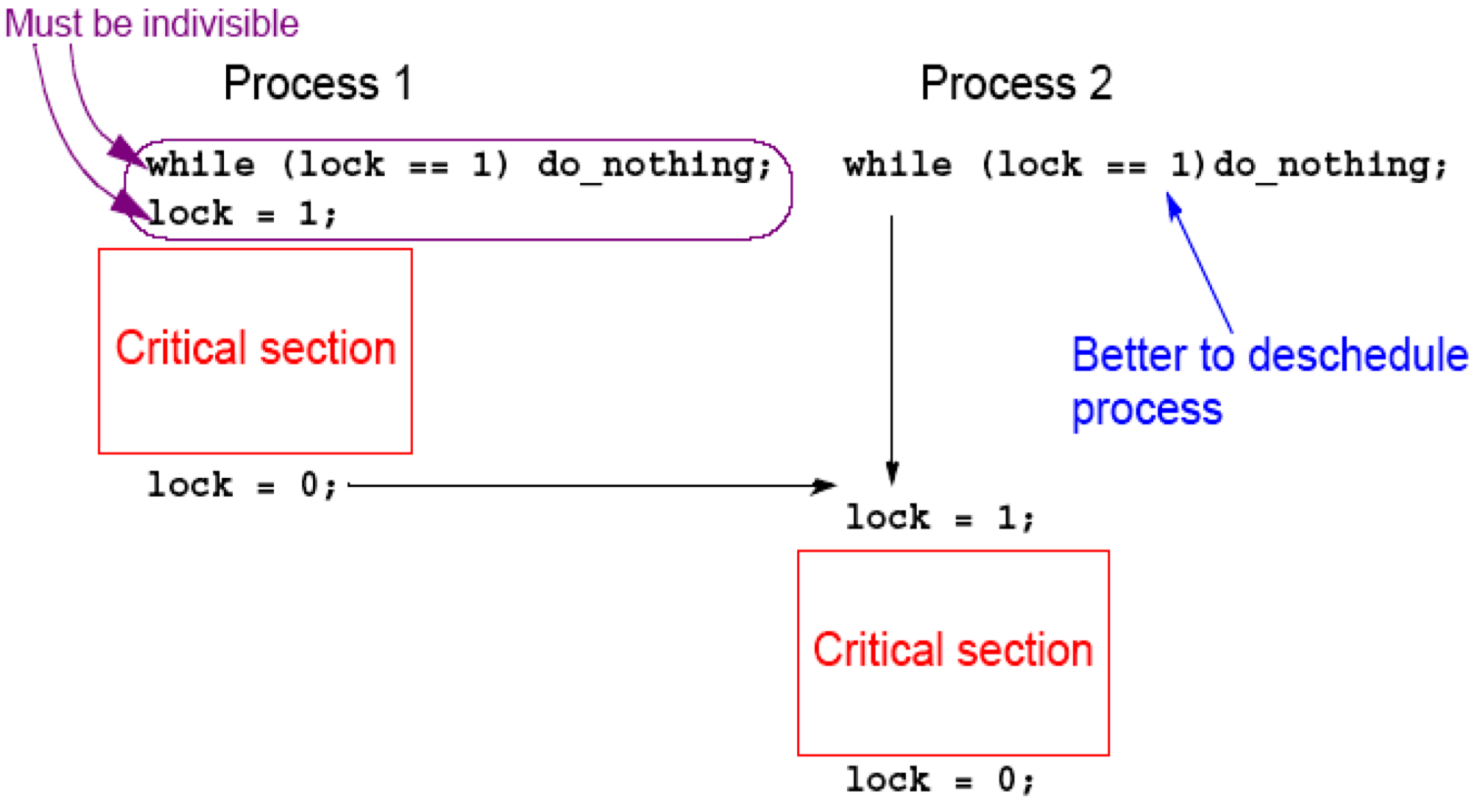
- Section of code to be executed by one process/thread at a time
  - Often because it accesses a shared resource or shared variable

- "Mutual Exclusion" is the term used to describe the mechanism used to ensure that only one process/thread is in a critical section at any time

- Several approaches for implementing mutual exclusion, such as:
  - Locks
  - Semaphores (Generalized locks)
  - Condition variables (allow threads to suspend until condition occurs)
  - Barriers

Locks

- Boolean variable (usually 1 bit) set to 1 when thread is in critical section, and 0 otherwise. Locks are initialized to 0.
- Before entering critical section, thread must "acquire" the lock, determine that it is 0, and set it to 1. All of this must be "atomic" (i.e., indivisible) so that multiple threads don't interfere with one another.



Pthreads: Mutex Lock Variables

Locks in Pthreads implemented using special mutex variables:

```
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
. . .
pthread_mutex_lock(&mutex1);
        <CRITICAL SECTION CODE IS HERE>
pthread_mutex_unlock(&mutex1);
```

When a thread finds that a mutex lock is locked, it waits until lock opens. If multiple threads reach the same lock, then OS eventually selects one to proceed. (No guarantees about when or which one or that the winning thread actually proceeds right away!) Only thread that locks a lock may unlock it.

Note the independence from the scheduler!

Better choice may be: `int pthread_mutex_trylock(pthread_mutex_t*)`

This either locks the mutex and returns 0, or returns EBUSY.

Semaphores

- A semaphore s is a non-negative integer variable on which only two operations are permitted:
  - P `[sem_wait]`: blocks until s > 0; then decrements s by 1 and returns
  - V `[sem_post]`: increments s by 1 and runs a waiting thread (if any)
- P and V operations are "atomic"; V is linked to the scheduler
- Implementation may unblock any thread that is already waiting
- Two types of semaphores:
  - Binary semaphores (values only 0 or 1): Used for locking critical sects.
  - Counting semaphores (records numbers of resources available or used): Applied to producer/consumer problems in operating systems
- Historical Notes:
  - Devised by Edsger Dijkstra in 1968
  - P comes from Dutch word "passeren," meaning "to pass"
  - V comes from Dutch work "vrijgeven," meaning "to release"

Semaphores vs. mutexes

Key differences:

- Semaphores aren't "owned" by any thread
- Semaphores may take on any unsigned integer value
- Semaphores may be initialized to any unsigned integer value
  - Often initialized to "fully unlocked" (1 for binary semaphore)
  - May be initialized to "locked" (0) by the main thread and then unlocked by other threads
- Any thread may execute *P* or *V* operation at any time (though care must be taken to ensure that semaphores take on expected values)
- Named semaphores available in Linux 2.6 and later
  - Uses a different set of semaphore functions (not discussed here)

Syntax of various semaphore functions

Semaphores are not part of Pthreads; you need to add this.

```
#include <semaphore.h>

int sem_init(
      sem_t*     semaphore_p   /* out */,
      int        shared        /* in  */,
      unsigned   initial_val   /* in  */);
```

Use 0 (or NULL) for multi-thread semaphores; use 1 for multi-process semaphores (not covered).

Can initialize to any unsigned int, so semaphore may start out "locked" or "unlocked".

```
int sem_destroy(sem_t*   semaphore_p   /* in/out */);
int sem_post(sem_t*      semaphore_p   /* in/out */);
int sem_wait(sem_t*      semaphore_p   /* in/out */);
```

Semaphores and Critical Sections

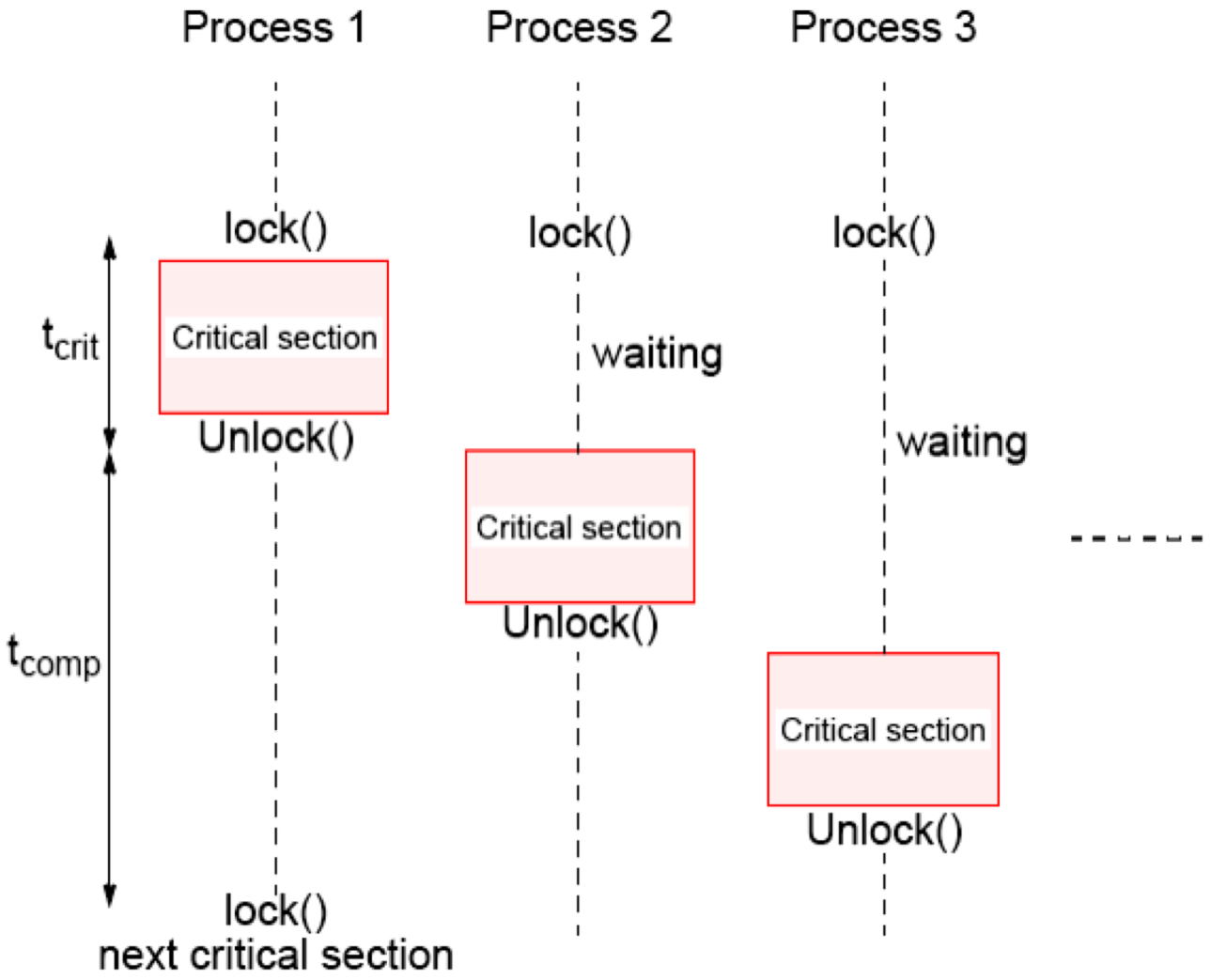
Mutual exclusion for a critical section can be achieved with a binary semaphore taking on only the values 0 and 1, since there is an implicit scheduling algorithm in the semaphore operations.

(A binary semaphore is like a lock tied to the thread scheduler.)

Example: Suppose s=1 initially (set using sem_init()). Then following works:

Thread 1	Thread 2	Thread 3
Noncritical Section	Noncritical Section	Noncritical Section
`sem_wait(&s)`	`sem_wait(&s)`	`sem_wait(&s)`
Critical Section	Critical Section	Critical Section
`sem_post(&s)`	`sem_post(&s)`	`sem_post(&s)`
Noncritical Section	Noncritical Section	Noncritical Section

Performance Issues with Critical Sections

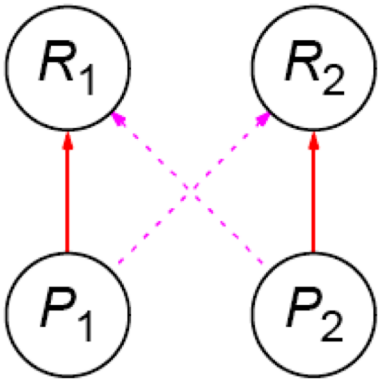


Concurrency Issues

- Deadlock: System can reach a state where no thread [in a group waiting for resources] changes state or makes any progress
- Coffman has described 4 conditions that are necessary and sufficient to permit deadlock:
  - Mutual Exclusion: Resources can only be used by one thread at a time
  - Hold & Wait: Threads already holding resources may request more
  - No Preemption: Resources may not be forcibly removed from a thread
  - Circular Wait: There can be a circular chain of threads in which each thread requires a resource held by the next thread in the chain
- Livelock is similar to deadlock, except that the states of the threads can change, although no progress is made
  - Example: two people meeting in a narrow hallway

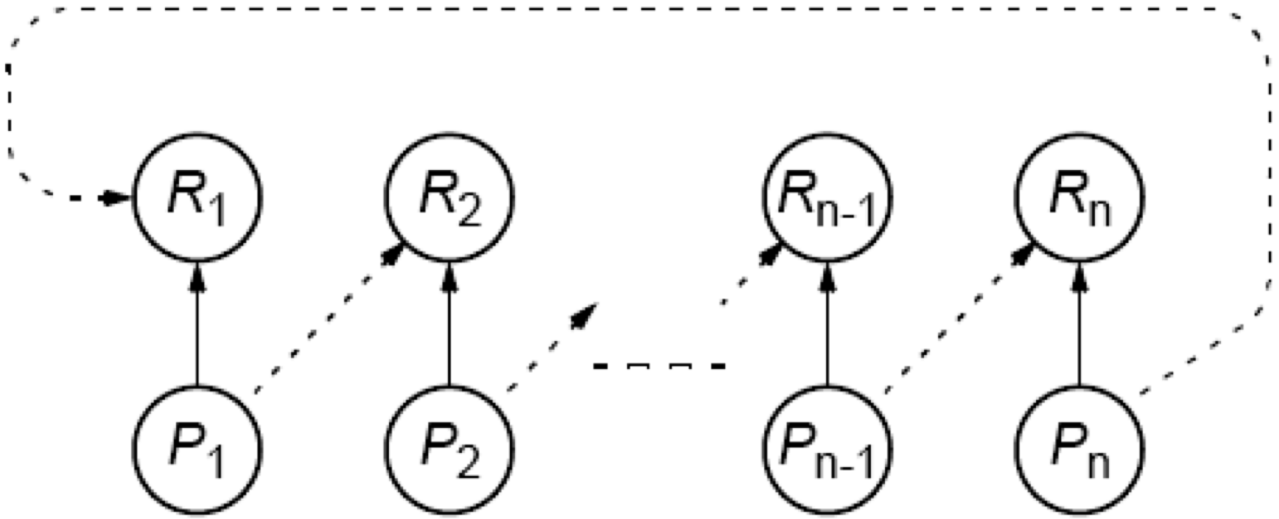
Deadlock

Need to avoid situation where processes hold “interlocking” resources in a way that causes deadlock or other problems.

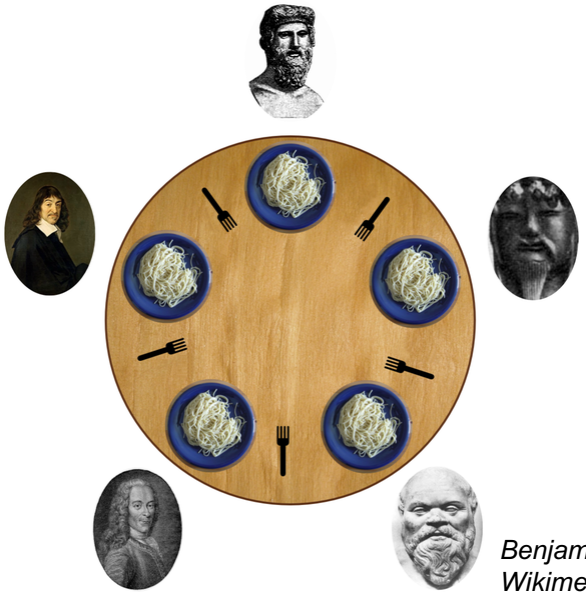


Resource

Process



Dining Philosophers Problem



Benjamin D. Esham for the Wikimedia Commons

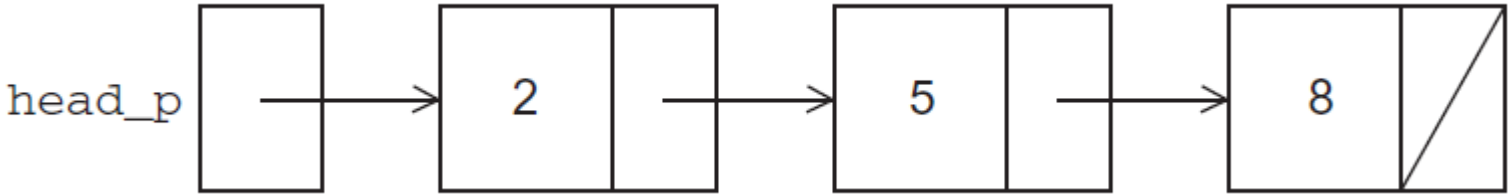
- Famous example that illustrates concurrency problems
  - Philosophers either "eat" or "think", but not both simultaneously
  - They require two forks to eat
  - How to ensure that no one starves?

- Possible solutions:
  - Simple timing rules:
    - With 1 fork, wait 2 mins for a 2nd; If no 2nd, put down 1st and wait 2 mins before trying again. (What's wrong with this?)
  - Central authority: "waiter" who grants permission to take a fork
  - Resource hierarchy: Take/release forks in order of "value"
  - Semaphores (= locks + scheduling)

Shared Data Structures

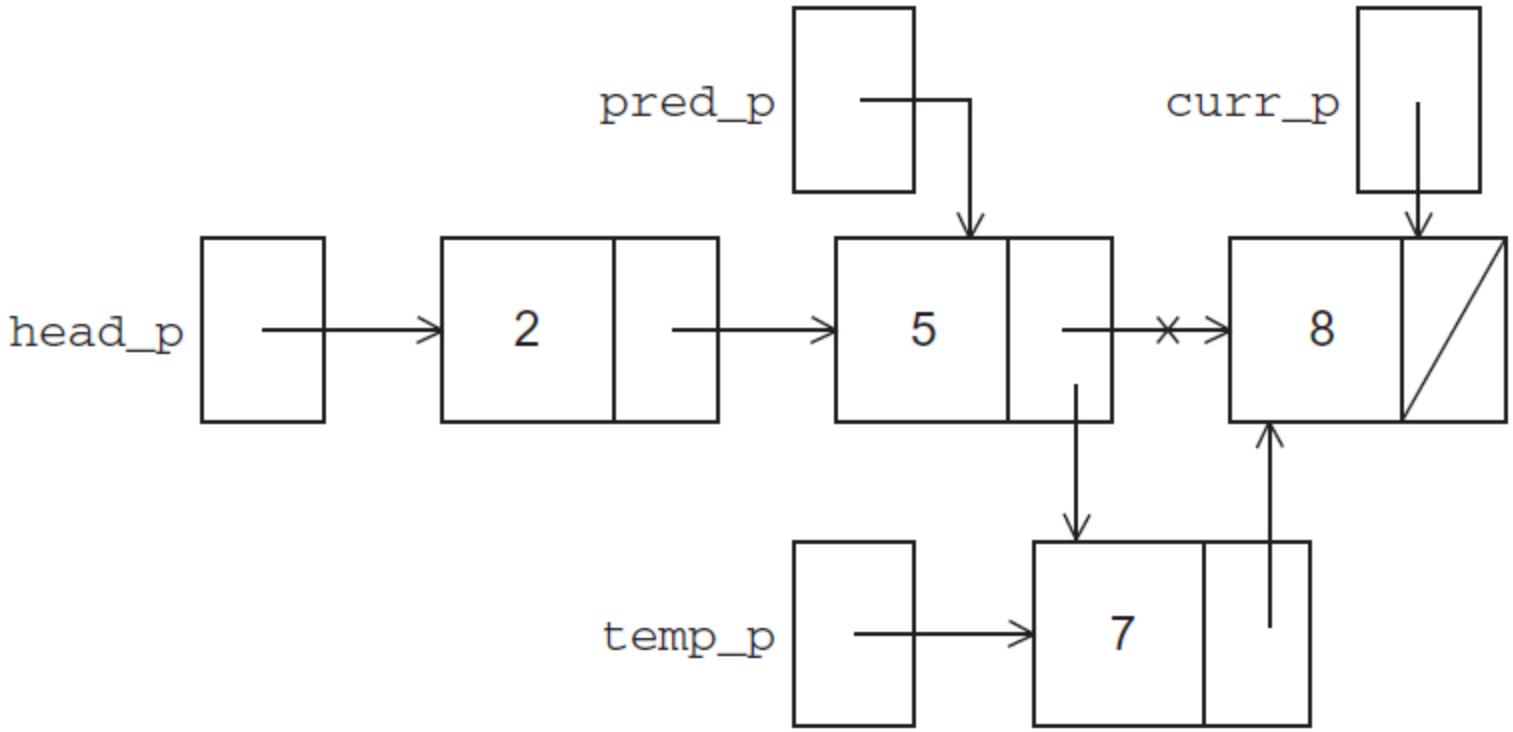
- Threaded programs often use shared, complex data structures that may be both read (often) and written/modified (infrequently)
  - Graphs
  - Stacks
  - Hash tables
  - Linked lists
- Linked List Example
  - Consider a shared linked list data structure containing sorted ints
  - Operations of interest:
    - *Member*: returns 1 if input is in the list; 0 otherwise. (Most frequent case)
    - *Insert*: inserts new entry in the list. (Relatively infrequent)
    - *Delete*: removes entry from the list. (Relatively infrequent)

Linked Lists




```
struct list_node_s {
    int data;
    struct list_node_s* next;
}
```

Inserting a new node into a list



Deleting a node from a linked list