Processes, Threads, and Synchronization

Lecture 02

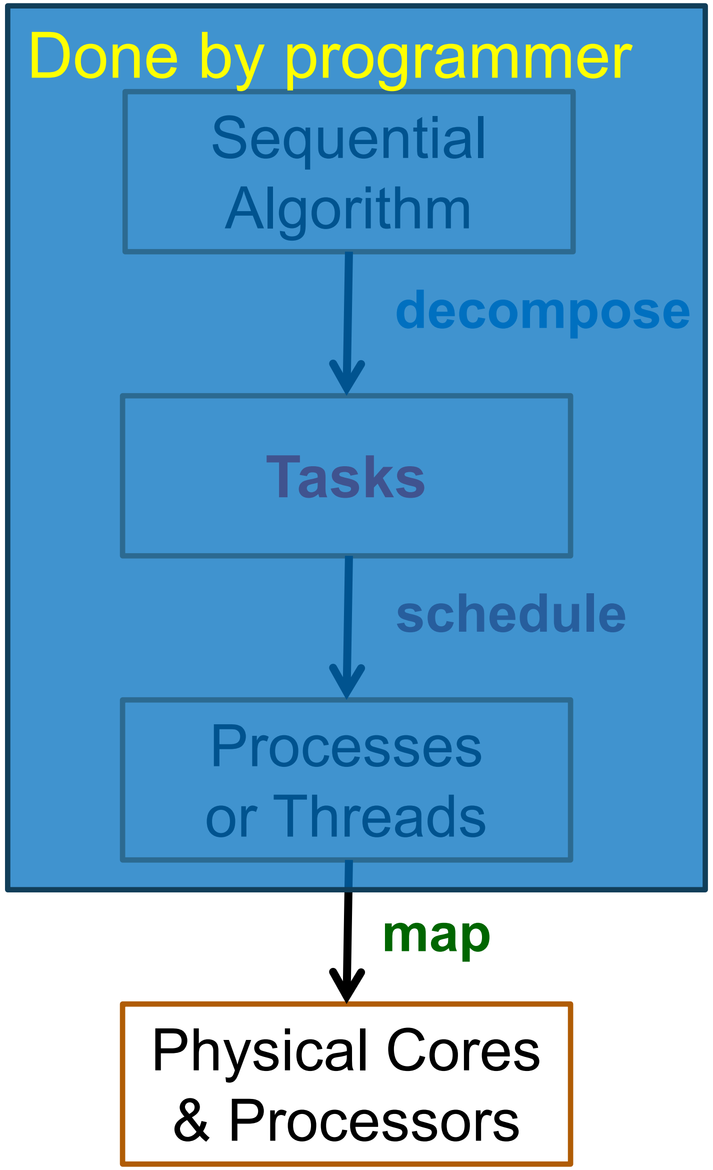
Program Parallelization: Steps

- 3 main steps:

  1. **Decomposition** of the computations

  2. **Scheduling** (assignment of tasks to processes (or threads))

  3. **Mapping** of processes (or threads) to physical processors (or cores)



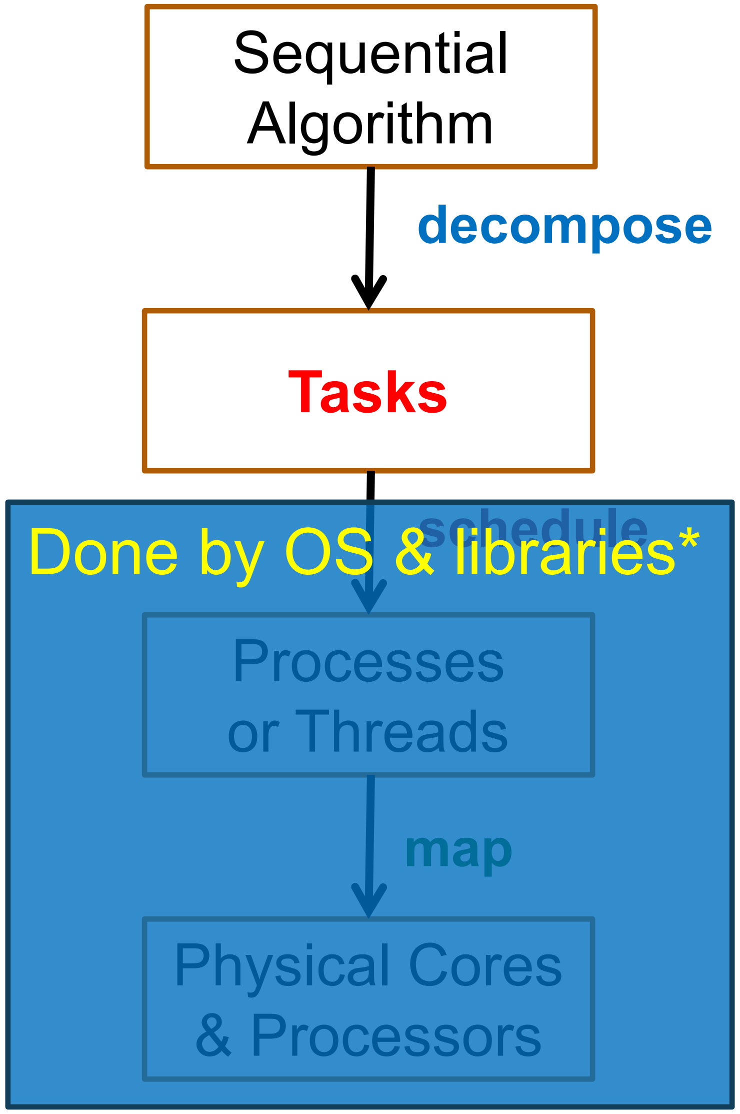
Program Parallelization: Steps

- 3 main steps:

  1. **Decomposition** of the computations

  2. **Scheduling** (assignment of tasks to processes (or threads))

  3. **Mapping** of processes (or threads) to physical processors (or cores)



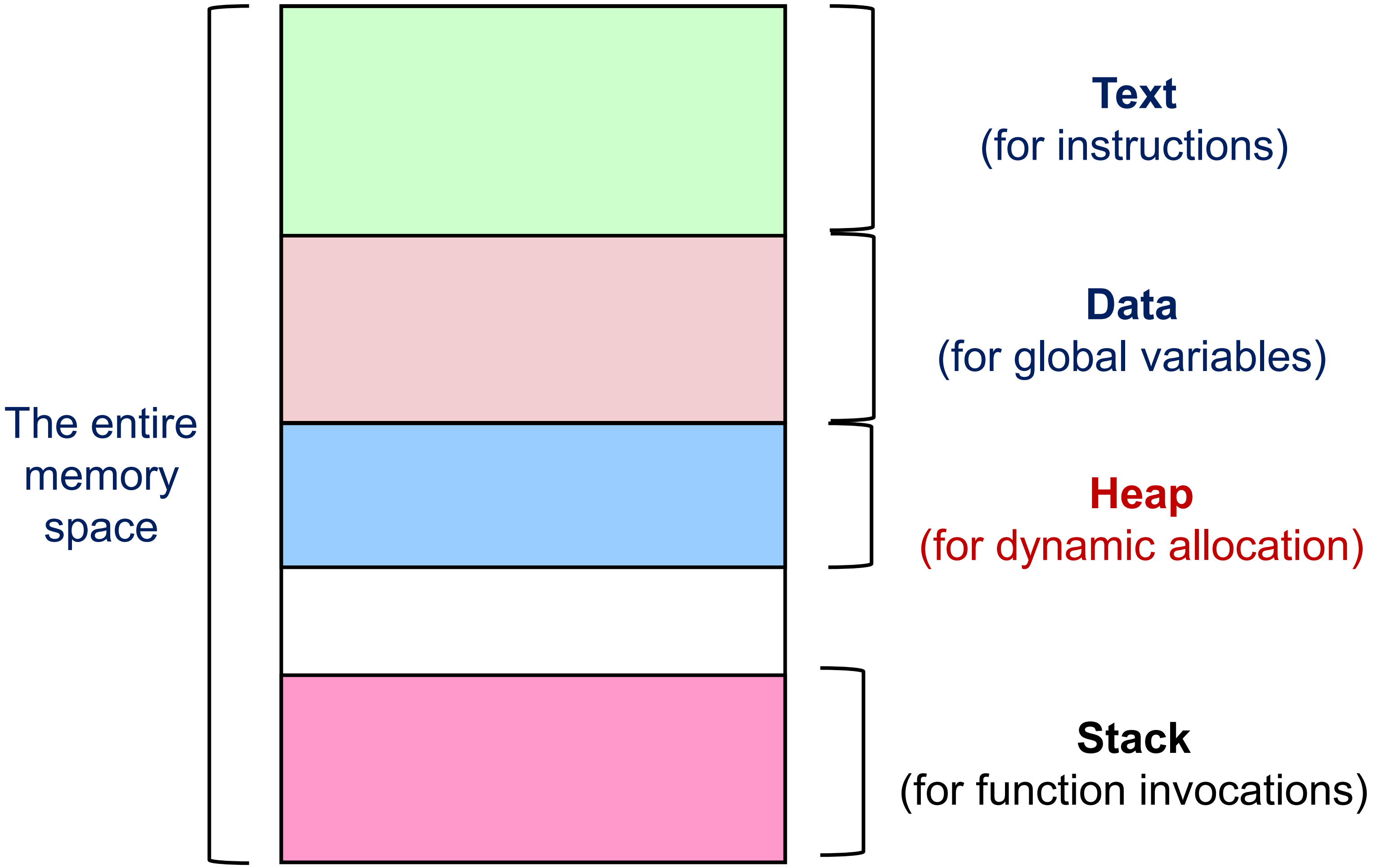
Abstractions of flow of control

PROCESSES AND THREADS

Processes

- A program in execution
  - Identified by PID (process ID)
  - Comprises:
    - executable program (PC),
    - global data
      - OS resources: open files, network connections
    - stack or heap
    - current values of the registers (GPRs and Special)
  - Own address space → exclusive access to its data
  - Two or more processes exchange data → need explicit communication

Memory Illustration of a Process



Multi-Programming (Multitasking)

- Several processes at different stages of execution
  - Need **context switch**, i.e., switching between processes
  - States of the suspended process must be saved ➔ overhead
  - 2 types of execution:
    - Time slicing execution – pseudo-parallelism
    - Parallel execution of processes on different resources

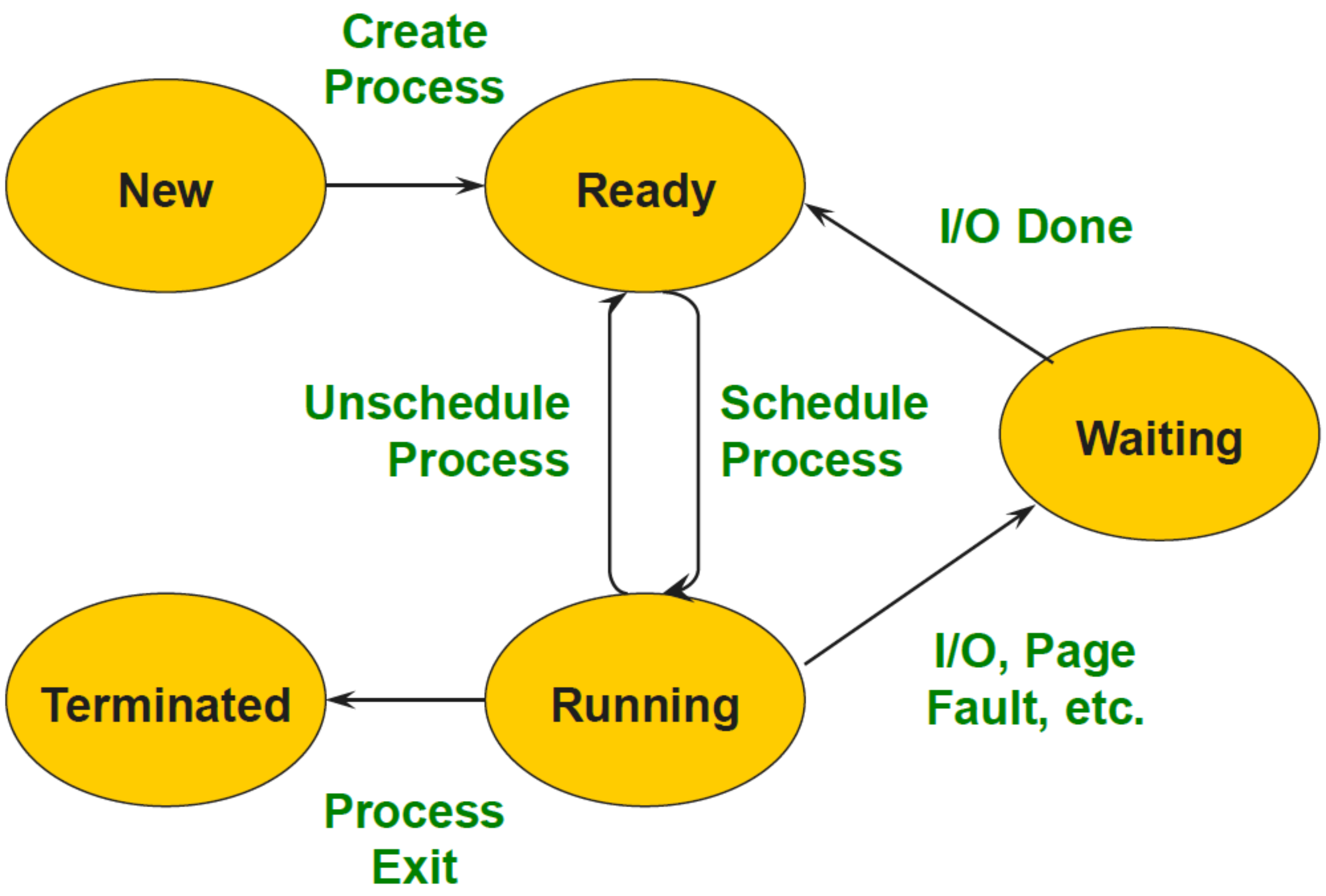
Create a New Process in Unix

- Process P_1 can create a new process P_2
  - **fork** system call
  - **int exec(char *prog, char *argv[])**
- P_2 is an identical copy of P_1 at the time of the fork call
  - P_2 works on a copy of the address space of P_1
  - P_2 executes the same program as P_1, starting with the instruction following the fork call
- P_2 gets its own process number
  - Use **ps** or **top** in Unix console to see a list of processes
- P_2 can execute different statements as P_1

Fork()

```
int main(int argc, char *argv[])
{
  char *name = argv[0];
  int child_pid = fork();
  if (child_pid == 0) {
    printf(“Child of %s is %d\n”, name, getpid());
    return 0;
  } else {
    printf(“My child is %d\n”, child_pid);
    return 0;
  }
}
```

Process State Graph



Why fork()?

- Very useful when the child…
  - Is cooperating with the parent
  - Relies upon the parent's data to accomplish its task
- Example: web server

```
while(1) {
  int sock =accept();if ((child_pid = fork()) == 0){
    Handle client request
  } else {
    Close socket
}}
```

Process Termination

- Use **exit(status)** in the child process
- Wait for a process in the parent process
  - **wait**
  - **waitpid (pid)**

Inter-process Communication (IPC)

- Cooperating processes have to share information
  - Shared memory
    - Need to protect access when reading/writing with locks
  - Message passing
    - Blocking & non-blocking
    - Synchronous & asynchronous
  - Unix specific:
    - Pipes & Signal

Process Interaction with OS

Exceptions

- Executing a **machine level instruction** can cause exception
- For example: Overflow, Underflow, Division by Zero, Illegal memory address, Mis-aligned memory access

Interrupts

- External events can interrupt the execution of a program
- Usually hardware related: Timer, Mouse Movement, Keyboard Pressed etc

- **Synchronous**
  - Occur due to program execution
- Have to execute an **exception handler**

- **Asynchronous**
  - Occur **independently** of program execution
- Have to execute an **interrupt handler**

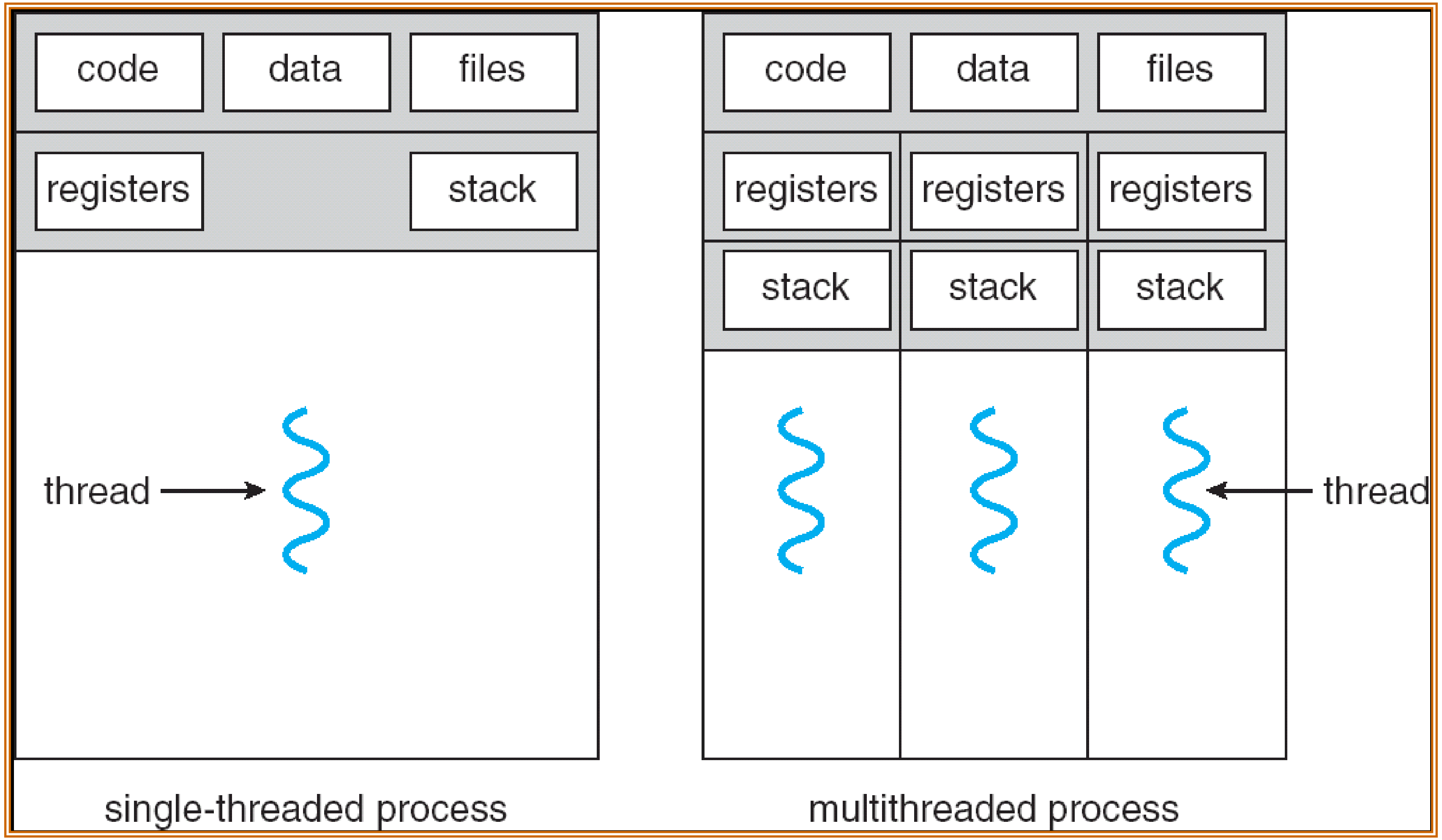
Disadvantages of Processes

- Creating a new process is costly
  - Overhead of system calls
  - All data structures must be allocated, initialized and copied
- Communicating between processes costly
  - Communication goes through the OS

Threads

- Extension of process model:
  - A process may consist of multiple independent control flows called **threads**
  - The thread defines a sequential execution stream within a process(PC, SP, registers)
- Threads share the address space of the process:
  - All threads belonging to the same process see the same value → **shared-memory architecture**

Process and thread: Illustration



- Taken from Operating System Concepts (7th Edition) by Silberschatz, Galvin & Gagne, published by Wiley

Threads (cont)

- Thread generation is faster than process generation
  - No copy of the address space is necessary

- Different threads of a process can be assigned run on different cores of a multicore processor

- 2 types of threads
  - User-level threads
  - Kernel threads

User-Level Threads

- Managed by a thread library – OS unaware of user-level threads so no OS support

- Advantages – switching thread context is fast

- Disadvantages
  - OS cannot map different threads of the same process to different execution resources → no parallelism
  - OS cannot switch to another thread if one thread executes a blocking I/O operation

Kernel Threads

- OS is aware of the existence of threads and can react correspondingly

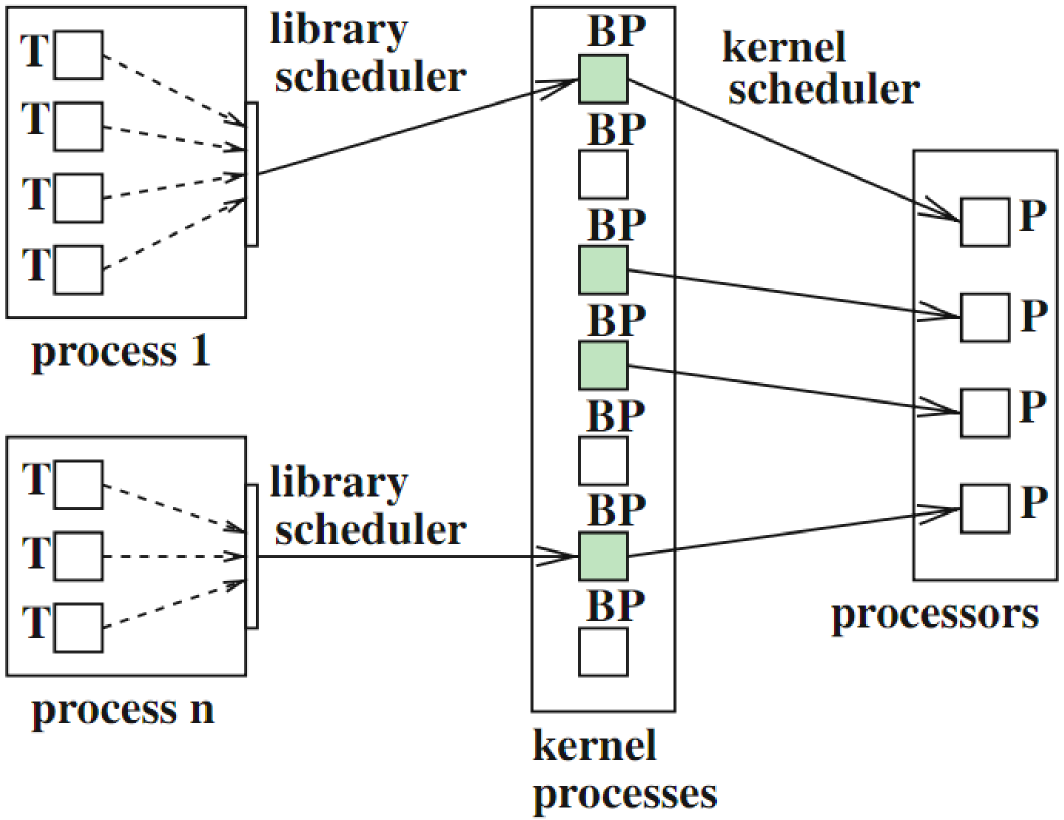
- Avoid disadvantages of user-level threads

- Efficient use of the cores in a multicore system

Many-to-One Mapping

- All user-level threads are mapped to one process

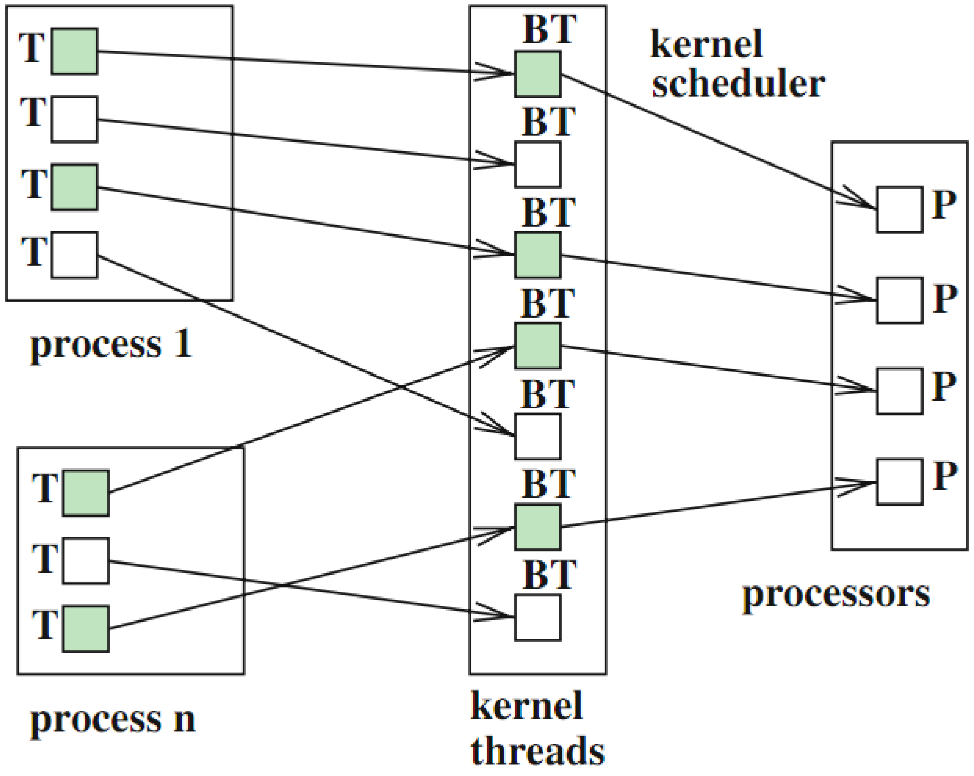
- Thread library is responsible for the scheduling of user-level threads



One-to-One Mapping

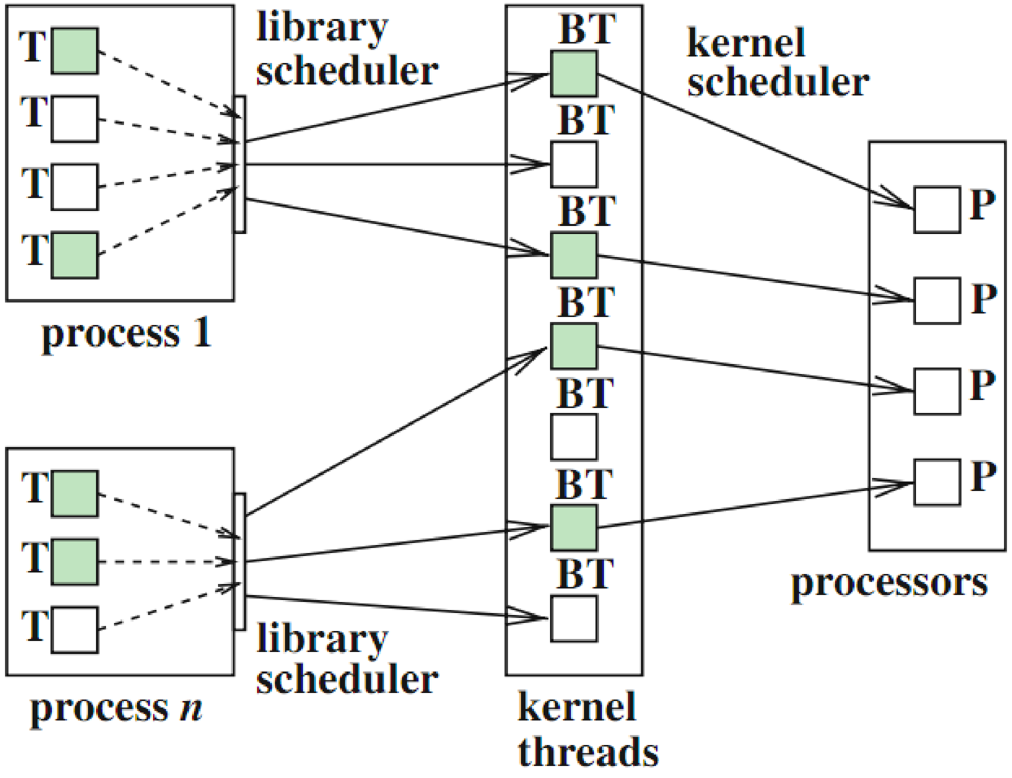
- Each user-level thread is assigned to exactly one kernel thread - no library scheduler needed

- OS is responsible for the scheduling and mapping of kernel threads



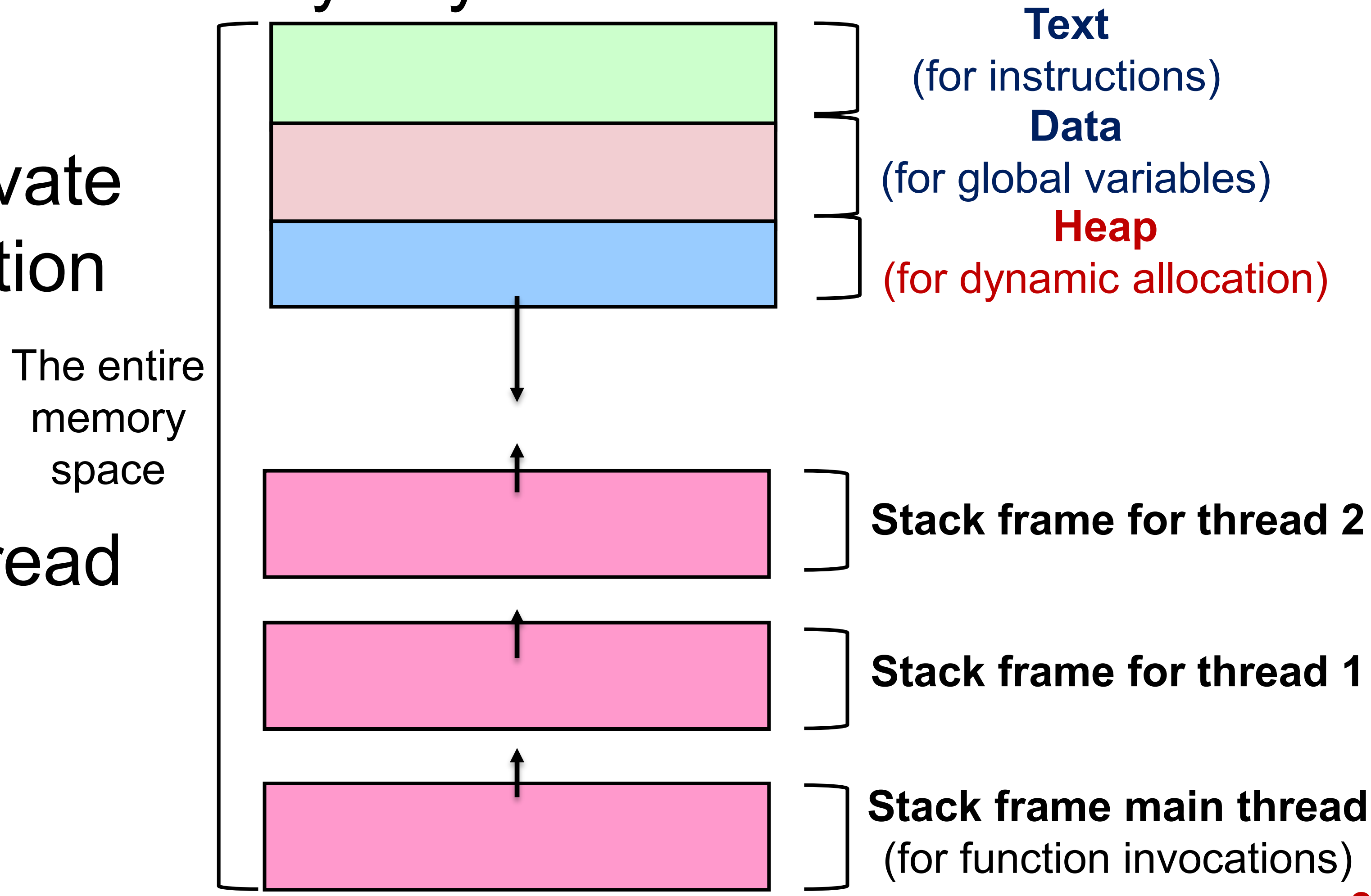
Many-to-Many Mapping

- Library scheduler assigns the user-level threads to a given set of kernel threads
- Kernel scheduler maps the kernel threads to the available execution resources
- At different points in time, a user thread may be mapped to a different kernel thread



Visibility of Data

- Global variables of a program and all dynamically allocated data objects can be accessed by any thread of this process
- Each thread has a private runtime stack for function stack frames
- Runtime stack of a thread exists iff the thread is active



POSIX Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );
main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;
    /* Create independent threads each of which will execute function */
     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */
     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}
void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

Number of Threads

- Number of threads should be
  - Suitable to parallelism degree of application
  - Suitable to available execution resources
  - Not be too large to keep the overhead for thread creation, management, and termination small

SYNCHRONIZATION

Introduction

- Threads cooperate in multithreaded programs
  - Share resources, access shared data structures
  - Coordinate their execution
    - One thread executes relative to another
- For correctness, control this cooperation
  - Threads interleave executions arbitrarily and at different rates
  - Scheduling is not under program control
- Use synchronization
  - Restrict the possible interleaving of thread executions

*Discuss in terms of threads, also applies to processes

Shared Resources

- Coordinating access to shared resources
  - Basic problem:
    - If two concurrent threads (processes) are accessing a shared variable, and that variable is read/ modified/ written by those threads, then access to the variable must be controlled to avoid erroneous behaviour
  - Mechanisms to control access to shared resources
    - Locks, mutexes, semaphores, monitors, condition variables, etc.
  - Patterns for coordinating accesses to shared resources
    - Bounded buffer, producer-consumer, etc.

Classic Example

- Implement a function to handle withdrawals from a bank account:

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    return balance;
}
```

- 2 people share a bank account with a balance of $1000
- Simultaneously withdraw $100 from the account

Classic Example - Threading

- Create a thread for each person to do the withdrawals
- These threads run on the same bank server:

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

- Possible problems?

Classic Example - Problem

- Execution of the two threads can be interleaved

Execution sequence seen by CPU

```
balance = get_balance(account);
balance = balance – amount;
```

```
balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
```

```
put_balance(account, balance);
```

Context switch

Race Condition

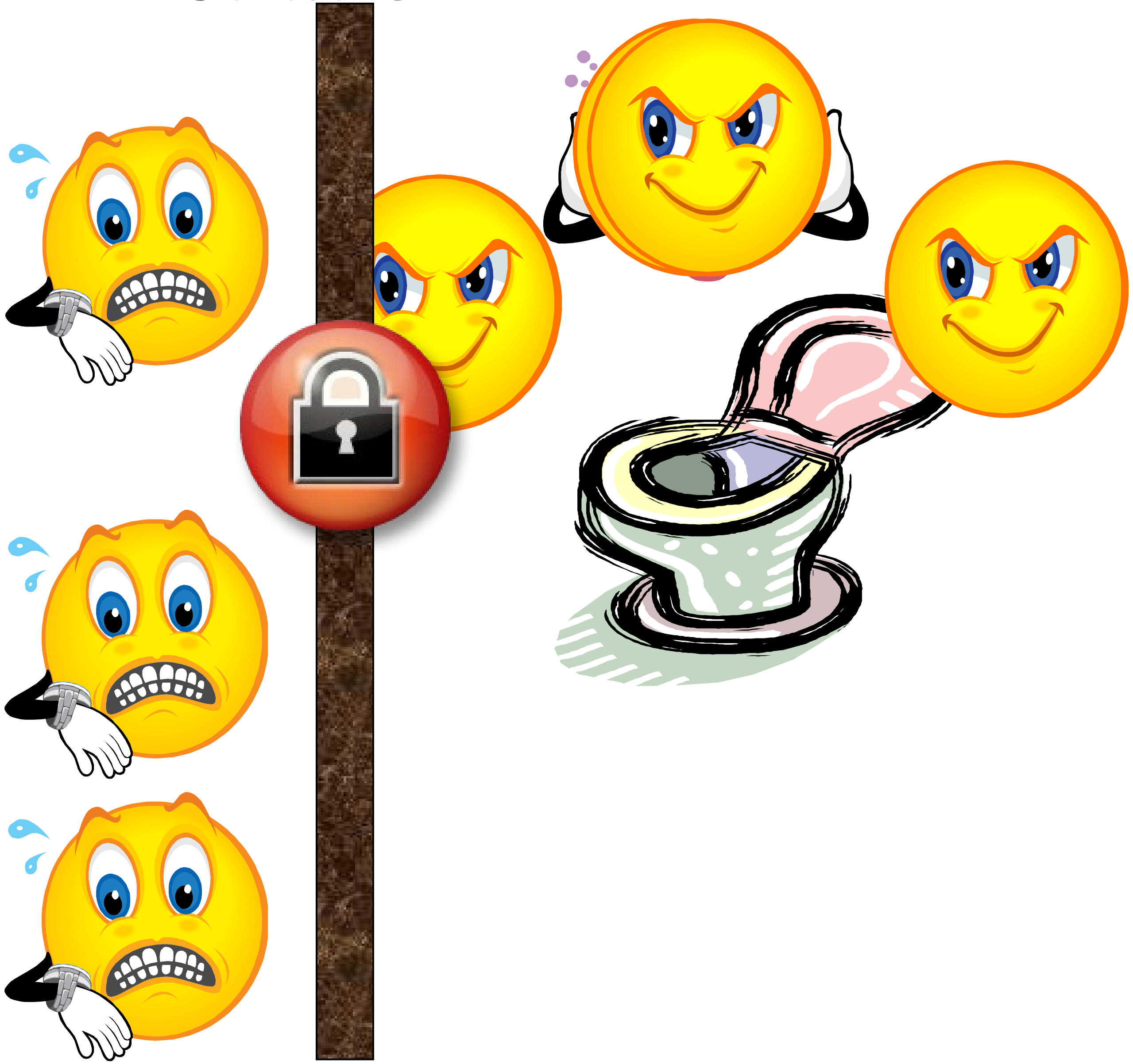
1. Two concurrent threads (or processes) access a shared resource without any synchronization

AND

2. At least one thread modifies the shared resource

- Solution: control access to these shared resources
- Necessary to synchronize access to any shared data structure
  - Buffers, queues, lists, hash tables, etc.

Critical Section: "Illustration"



Mutual Exclusion

- Use mutual exclusion to synchronize access to shared resources
  - This allows us to have large atomic blocks
- Code sequence that uses mutual exclusion is called critical section
  - Only one thread at a time can execute in the critical section
  - All other threads have to wait on entry
  - When a thread leaves a critical section, another can enter

Mechanisms

- Locks
  - Primitive, minimal semantics, used to build others
- Semaphores
  - Basic, easy to get the hang of, but hard to program with
- Monitors
  - High-level, requires language support, operations implicit
- Messages
  - Simple model of communication and synchronization based on atomic transfer of data across a channel
  - Direct application to distributed systems
  - Messages for synchronization are straightforward (once we see how the others work)

Locks

- Two operations
  - acquire(): to enter a critical section
  - release(): to leave a critical section
- Pair calls to acquire and release
  - Between acquire/release, the thread holds the lock
  - Acquire does not return until any previous holder releases
  - What can happen if the calls are not paired?
- Locks can spin (a spinlock) or block (a mutex)

Using Locks

```
withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}
```

Critical Section

```
acquire(lock);
balance = get_balance(account);
balance = balance – amount;
```

```
acquire(lock);
```

```
put_balance(account, balance);
release(lock);
```

```
balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
release(lock);
```

Semaphores

- Semaphores are an abstract data type that provide mutual exclusion through atomic counters
  - Described by Dijkstra in the “THE” system in 1968
- Semaphores are “integers” that support two operations:
  - Semaphore::Wait(): decrement, block until semaphore is open
    - Also P(), after the Dutch word for “try to reduce” (down)
  - Semaphore::Signal(): increment, allow another thread to enter
    - Also V() after the Dutch word for increment (up)
  - Semaphore safety property: the semaphore value is always greater than or equal to 0

Semaphore Types

- Mutex semaphore (or binary semaphore)
  - Represents single access to a resource
  - Guarantees mutual exclusion to a critical section
- Counting semaphore (or general semaphore)
  - Multiple threads can pass the semaphore
  - Number of threads determined by the semaphore "count"
    - mutex has count = 1, counting has count = N

Example

```
struct Semaphore {
    int value;
    Queue q;
} S;
withdraw (account, amount) {
    wait(S);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    signal(S);
    return balance;
}
```

critical section

Threads block

It is undefined which thread runs after a signal

```
wait(S);
balance = get_balance(account);
balance = balance – amount;
```

```
wait(S);
```

```
wait(S);
```

```
put_balance(account, balance);
signal(S);
```

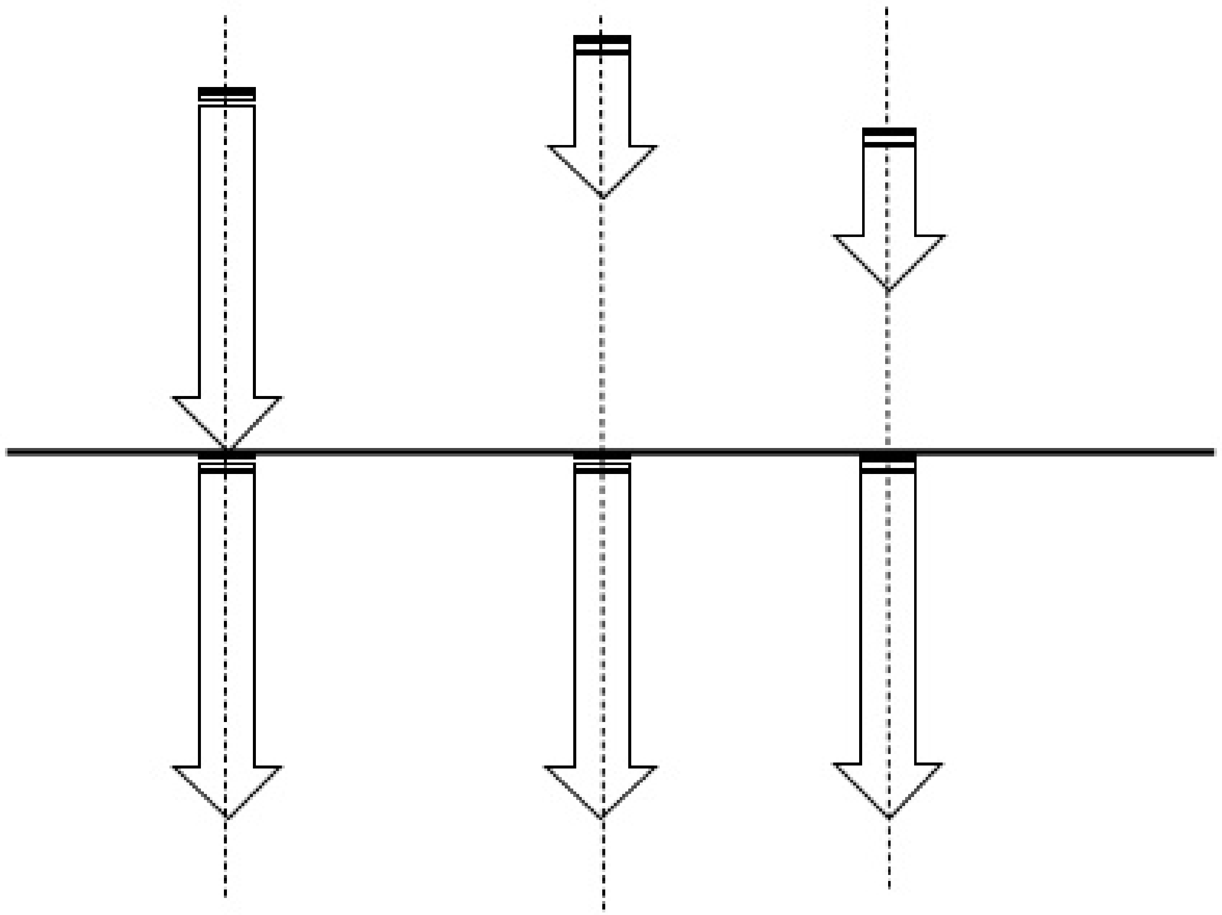
```
…
signal(S);
```

```
…
signal(S);
```

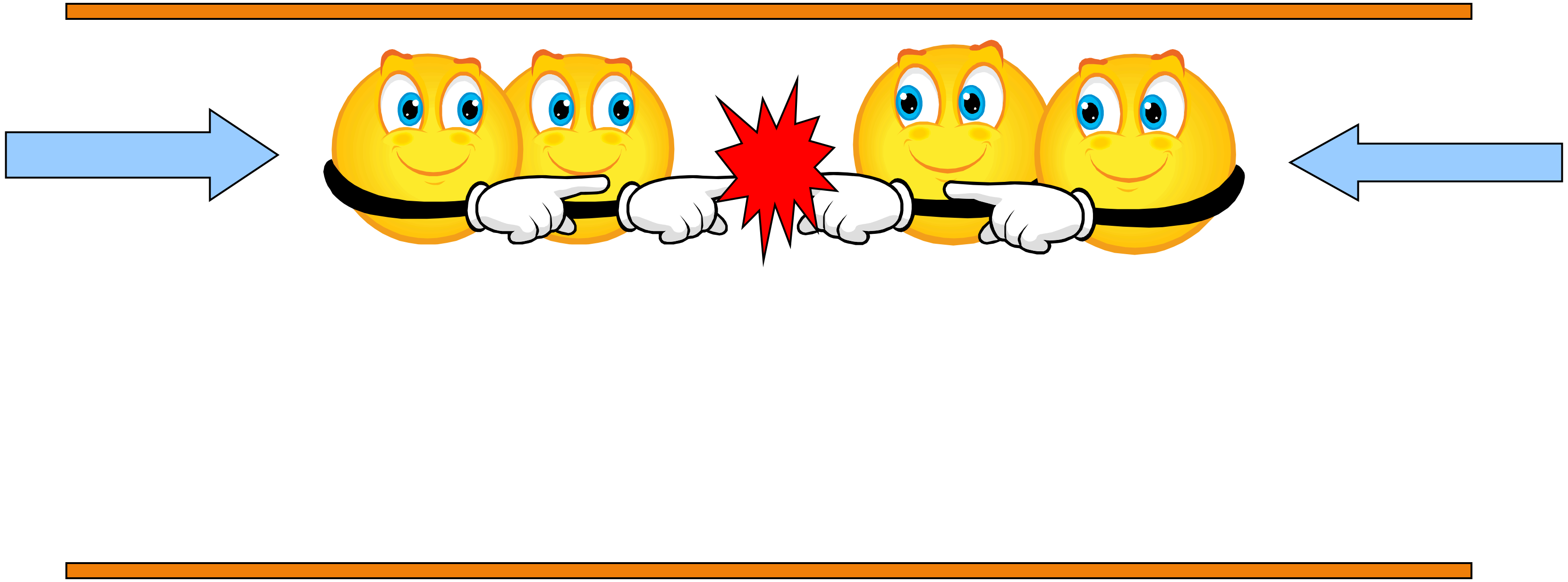
Semaphores Summary

- Semaphores can be used as a mutex
- However, they have some drawbacks
  - They are essentially shared global variables
    - Can potentially be accessed anywhere in program
  - No connection between the semaphore and the data being controlled by the semaphore
  - Used both for critical sections (mutual exclusion) and coordination (scheduling)
- Sometimes hard to use and prone to bugs

Barrier



Deadlock



Deadlock

- Definition:
  - Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set.
- Deadlock is a problem that can arise:
  - When processes compete for access to limited resources
  - When processes are incorrectly synchronized

Condition for Deadlock

- Deadlock can exist if and only if the following four conditions hold simultaneously:
  1. Mutual exclusion – At least one resource must be held in a non-sharable mode
  2. Hold and wait – There must be one process holding one resource and waiting for another resource
  3. No pre-emption – Resources cannot be pre-empted (critical sections cannot be aborted externally)
  4. Circular wait – There must exist a set of processes [P1, P2, P3,…,Pn] such that P1 is waiting for P2, P2 for P3, etc.

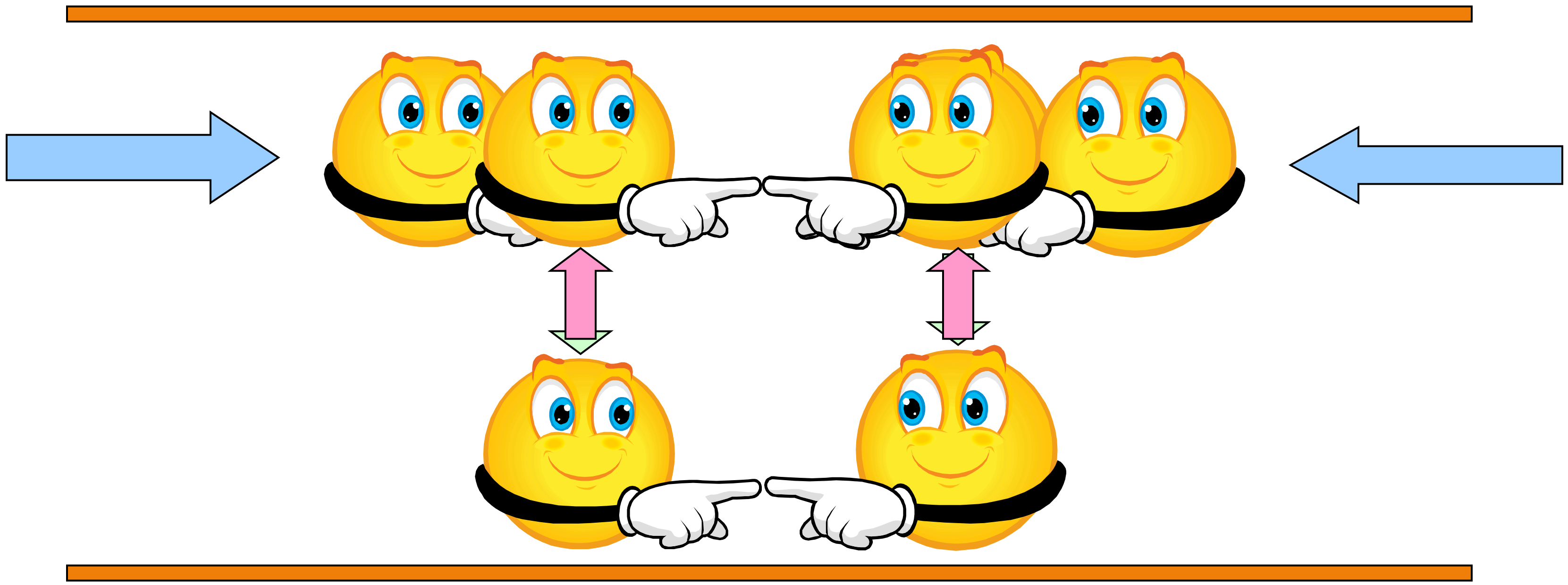
Dealing with Deadlock

- There are four approaches for dealing with deadlock:
  - Ignore it–how lucky do you feel?
  - Prevention–make it impossible for deadlock to happen
  - Avoidance–control allocation of resources
  - Detection and recovery–look for a cycle in dependencies

Starvation

- Starvation is a situation where a process is prevented from making progress because some other process has the resource it requires
- Starvation is a side effect of the scheduling algorithm
  - OS: A high priority process always prevents a low priority process from running on the CPU
  - One thread always beats another when acquiring a lock

Livelock



CLASSICAL SYNCHRONIZATION PROBLEMS

Classic Synchronization Problems

- Producer-consumer
  - Infinite buffer
  - Finite buffer
- Readers-writers
- Dining philosophers
- Barbershop
- …

Producer-consumer

- Producers create items of some kind and add them to a data structure
- Consumers remove the items and process them
- Variables:
  - mutex = Semaphore (1)
  - items = Semaphore (0)

Producer-consumer

Producer

- event = waitForEvent ()
- mutex.wait ()
  - buffer.add (event)
  - items.signal ()
- mutex.signal ()

Consumer

- items.wait ()
- mutex.wait ()
  - event = buffer.get ()
- mutex.signal ()
- event.process ()

Improved Producer-consumer

Producer

- event = waitForEvent ()
- mutex.wait ()
  - buffer.add (event)
- mutex.signal ()
- items.signal ()

Consumer

- items.wait ()
- mutex.wait ()
  - event = buffer.get ()
- mutex.signal ()
- event.process ()

Improved Producer-consumer

Producer

- event = waitForEvent ()
- mutex.wait ()
  - buffer.add (event)
- mutex.signal ()
- items.signal ()

Consumer

- items.wait ()
- mutex.wait ()
  - event = buffer.get ()
- mutex.signal ()
- event.process ()

Broken Producer-consumer

Producer

- event = waitForEvent ()
- mutex.wait ()
  - buffer.add (event)
- mutex.signal ()
- items.signal ()

Consumer

- mutex.wait ()
  - items.wait ()
  - event = buffer.get ()
- mutex.signal ()
- event.process ()

Producer-consumer with Finite Buffer

Producer

- event = waitForEvent ()
- spaces.wait ()
- mutex.wait ()
  - buffer.add (event)
- mutex.signal ()
- items.signal ()

Consumer

- items.wait ()
- mutex.wait ()
  - event = buffer.get ()
- mutex.signal ()
- spaces.signal ()
- event.process ()

Readers-writers Problem

- Any number of readers can be in the critical section simultaneously
- Writers must have exclusive access to the critical section
- Variables:
  - int readers = 0
  - mutex = Semaphore (1)
  - roomEmpty = Semaphore (1)

Readers-writers

Writers

- roomEmpty.wait ()
  - #critical section for writers
- roomEmpty.signal ()

Readers

- mutex.wait ()
  - readers += 1
  - if readers == 1:
    - roomEmpty.wait () # first in locks
- mutex.signal ()
- # critical section for readers
- mutex.wait ()
  - readers -= 1
  - if readers == 0:
    - roomEmpty.signal () # last out unlocks
- mutex.signal ()

Lightswitch Definition

class Lightswitch :

- def __init__ (self):
  - self.counter = 0
  - self.mutex = Semaphore (1)
- def lock (self , semaphore):
  - self.mutex.wait ()
    - self.counter += 1
    - if self.counter == 1:
      - semaphore.wait ()
    - self.mutex.signal ()
- def unlock (self , semaphore):
  - self.mutex.wait ()
    - self.counter -= 1
    - if self.counter == 0:
      - semaphore.signal ()
  - self.mutex.signal ()

Readers-writers with Lightswitch

Writers

- roomEmpty.wait ()
  - #critical section for writers
- roomEmpty.signal ()

Readers

- readLightswitch.lock (roomEmpty)
  - # critical section
- readLightswitch.unlock (roomEmpty)

- #starving writers
- Use a
  - turnstile = Semaphore (1)

No-starve Readers-writers

Writers

- turnstile.wait ()
  - roomEmpty.wait ()
    - # critical section for writers
- turnstile.signal ()
- roomEmpty.signal ()

Readers

- turnstile.wait ()
- turnstile.signal ()
- readSwitch.lock (roomEmpty)
  - # critical section for readers
- readSwitch.unlock (roomEmpty)

Readers-writers with Priorities

Writers

- writeSwitch.lock (noReaders)
  - noWriters.wait ()
  - # critical section for writers
  - noWriters.signal()
- writeSwitch.unlock (noReaders)

Readers

- noReaders.wait ()
  - readSwitch.lock (noWriters)
- noReaders.signal ()
- # critical section for readers
- readSwitch.unlock (noWriters)

Readings

- Any operating system textbook; CS2106
- CSE 120: Principles of Computer Operating Systems, UCSD, http://cseweb.ucsd.edu/classes/fa16/cse120-a/
- The Little Book of Semaphores by Allen Downey, hsttp://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf

LOCK IMPLEMENTATION

Implementing Locks (1)

- An attempt:

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock→held);
    lock→held = 1;
}
void release (lock) {
    lock→held = 0;
}
```

busy-wait (spin-wait) for lock to be released

- This is called a spinlock because a thread spins waiting for the lock to be released
- Does this work?

Implementing Locks (2)

- No. Two independent threads may both notice that a lock has been released and thereby acquire it.

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (lock→held);
    lock→held = 1;
}
void release (lock) {
    lock→held = 0;
}
```

A context switch can occur here, causing a race condition

Implementing Locks (3)

- The problem: implementation of locks has critical sections, too
  - How do we stop the recursion?
- The implementation of acquire/release must be atomic
  - An atomic operation is one which executes as though it could not be interrupted
  - Code that executes “all or nothing”
- Need help from hardware
  - Atomic instructions (e.g., test-and-set)
  - Disable/enable interrupts (prevents context switches)

Atomic Instructions: Test-and-set

- The semantics of test-and-set are:
  - Record the old value
  - Set the value to indicate available
  - Return the old value
- Hardware executes it atomically!

```
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

Lock with Test-and-set

```
struct lock {
    int held = 0;
}
void acquire (lock) {
    while (test-and-set(&lock→held));
}
void release (lock) {
    lock→held = 0;
}
```

Problems with Spinlocks

- Spinlocks are wasteful
  - If a thread is spinning on a lock, then the thread holding the lock cannot make progress (on a uniprocessor)
- How did the lock holder give up the CPU in the first place?
  - Lock holder calls yield or sleep
  - Involuntary context switch

Higher-level Synchronization

- All synchronization requires atomicity
  - Use "atomic"
- Look at two common high-level mechanisms
  - Semaphores: binary (mutex) and counting
  - Monitors: mutexes and condition variables locks as primitives