

# **Part 2: Processes & Threads**

Part 1: Processes & Scheduling

# Processes and threads

---

- ✦ Processes
- ✦ Threads
- ✦ Scheduling
- ✦ Interprocess communication
- ✦ Classical IPC problems

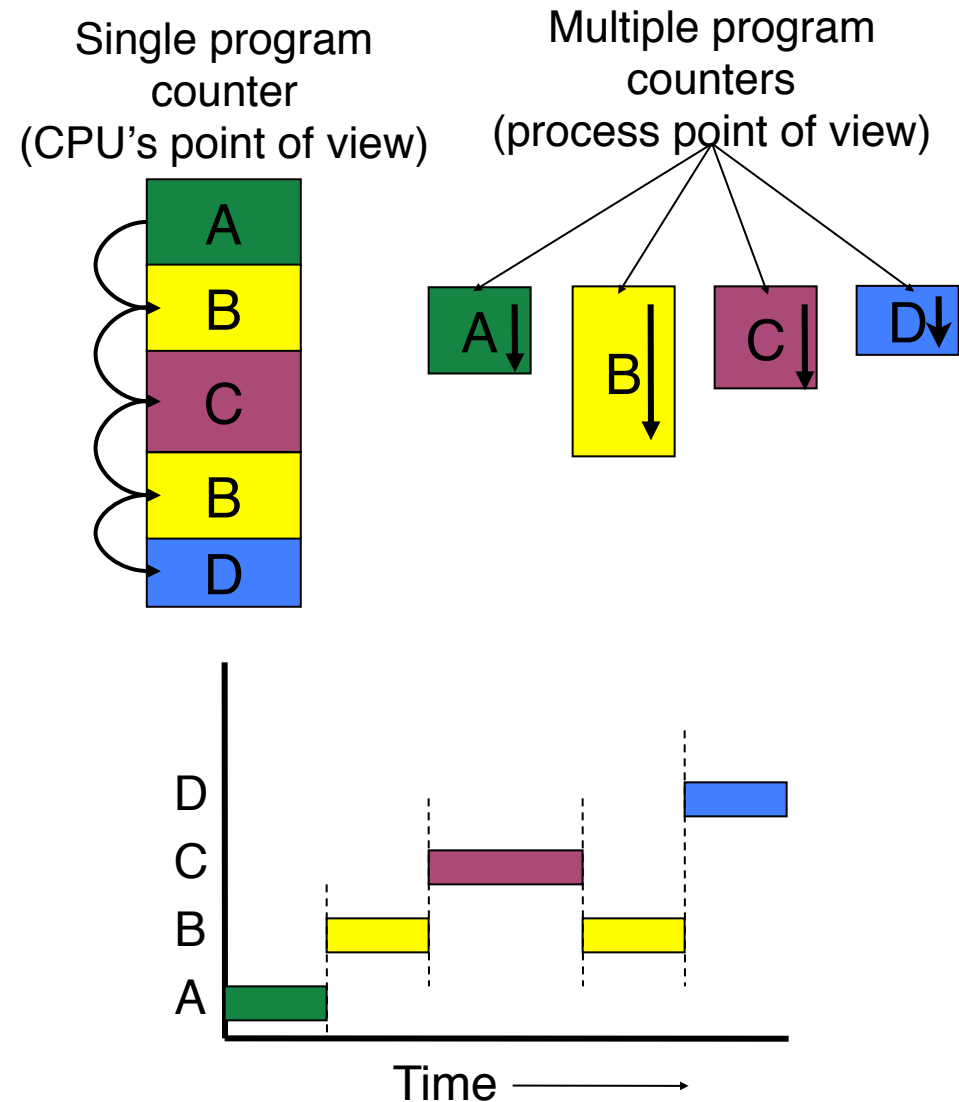
# What is a process?

---

- ✦ Code, data, and stack
  - Usually (but not always) has its own address space
- ✦ Program state
  - CPU registers
  - Program counter (current location in the code)
  - Stack pointer
- ✦ Only one process can be running in the CPU at any given time!

# The process model

- ✦ Multiprogramming of four programs
- ✦ Conceptual model
  - 4 independent processes
  - Processes run sequentially
- ✦ Only one program active at any instant!
  - That instant can be very short...
  - Only applies if there's a single CPU in the system



# When is a process created?

---

- ✦ Processes can be created in two ways
  - System **initialization**: one or more processes created when the OS starts up
  - **Execution** of a process creation system call: something explicitly asks for a new process
- ✦ System calls can come from
  - User request to create a new process (system call executed from user shell)
  - Already running processes
    - User programs
    - System daemons

# When do processes end?

---

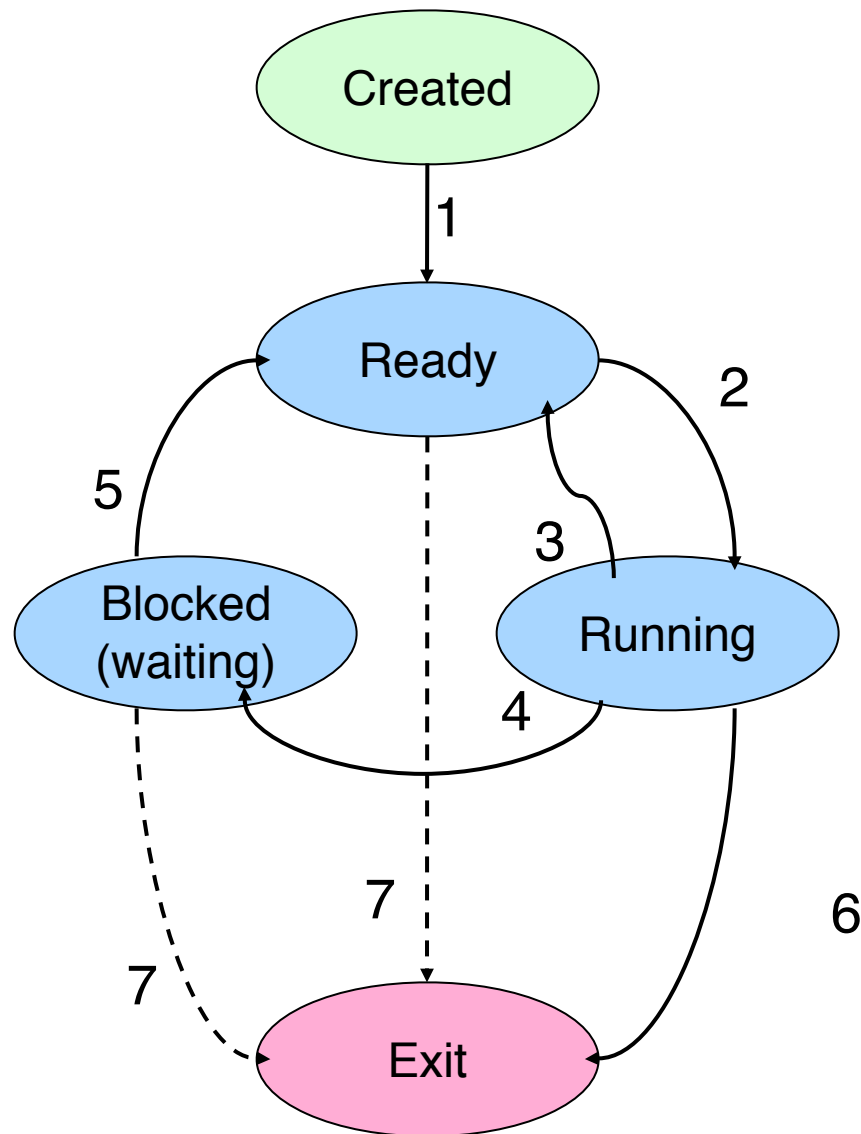
- ✦ Conditions that **terminate** processes can be
  - Voluntary
  - Involuntary
- ✦ Voluntary
  - Normal exit
  - Error exit
- ✦ Involuntary
  - Fatal error (only sort of involuntary)
  - Killed by another process

# Process hierarchies

---

- ✦ Parent creates a child process
  - Child processes can create their own children
- ✦ Forms a hierarchy
  - UNIX calls this a “process group”
  - If a process terminates, its children are “inherited” by the terminating process’s parent
- ✦ Windows has process groups
  - Multiple processes grouped together
  - One process is the “group leader”

# Process states



## ✦ Process in one of 5 states

- Created
- Ready
- Running
- Blocked
- Exit

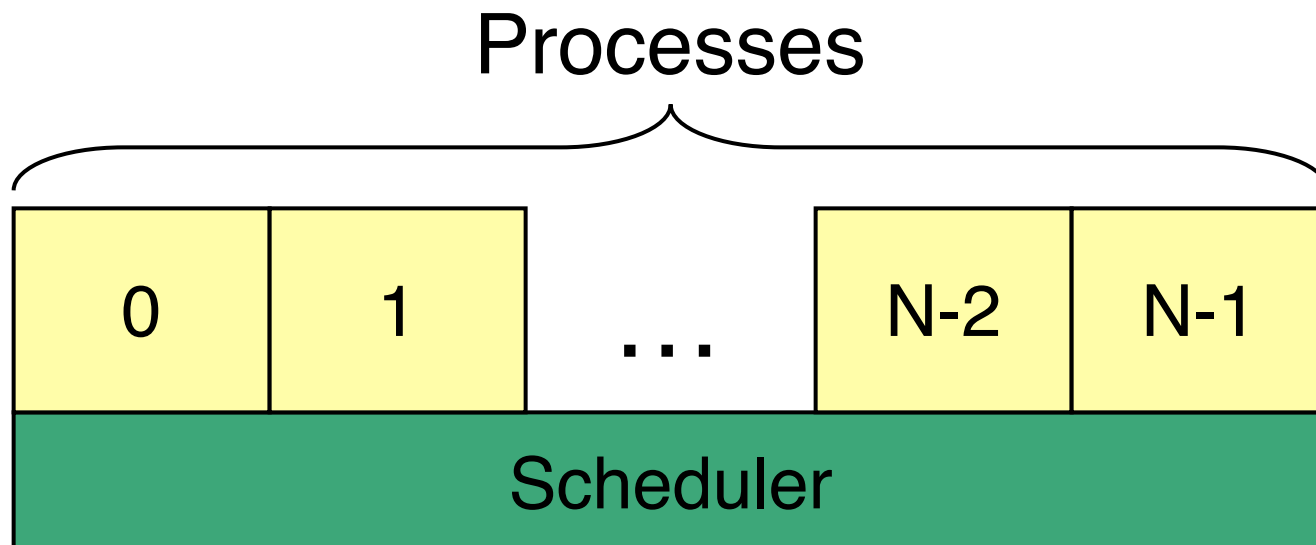
## ✦ Transitions between states

- Process enters ready queue
- Scheduler picks this process
- Scheduler picks a different process
- Process waits for event (such as I/O)
- Event occurs
- Process exits
- Process ended by another process

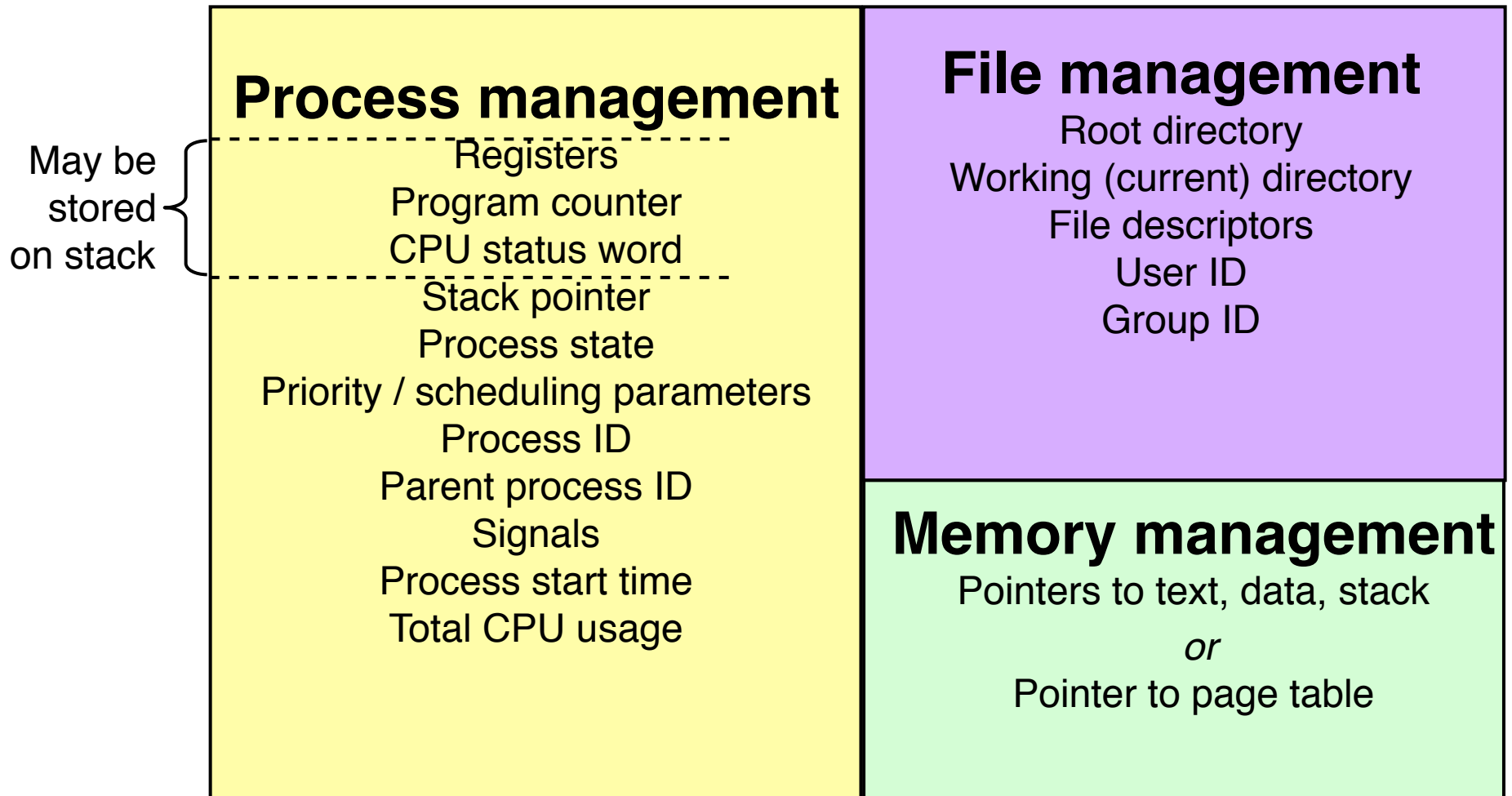


# Processes in the OS

- ◆ Two “layers” for processes
- ◆ Lowest layer of process-structured OS handles interrupts, scheduling
- ◆ Above that layer are sequential processes
  - Processes tracked in the process table
  - Each process has a process table entry



# What's in a process table entry?



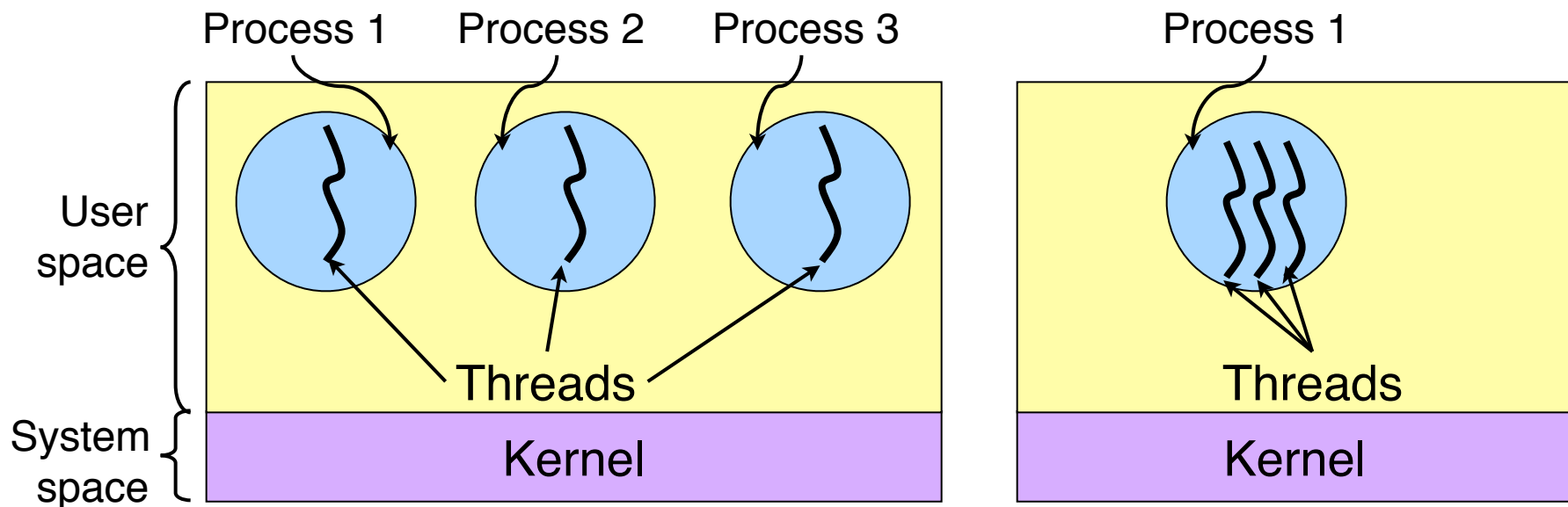
# What happens on a trap/ interrupt?

---

1. Hardware saves program counter (on stack or in a special register)
2. Hardware loads new PC, identifies interrupt
3. Assembly language routine saves registers
4. Assembly language routine sets up stack
5. Assembly language calls C to run service routine
6. Service routine calls scheduler
7. Scheduler selects a process to run next (might be the one interrupted...)
8. Assembly language routine loads PC & registers for the selected process

# Threads: “processes” sharing memory

- ◆ Process == address space
- ◆ Thread == program counter / stream of instructions
- ◆ Two examples
  - Three processes, each with one thread
  - One process with three threads



# Process & thread information

## Per process items

Address space  
Open files  
Child processes  
Signals & handlers  
Accounting info  
*Global variables*

## Per thread items

Program counter  
Registers  
Stack & stack pointer  
State

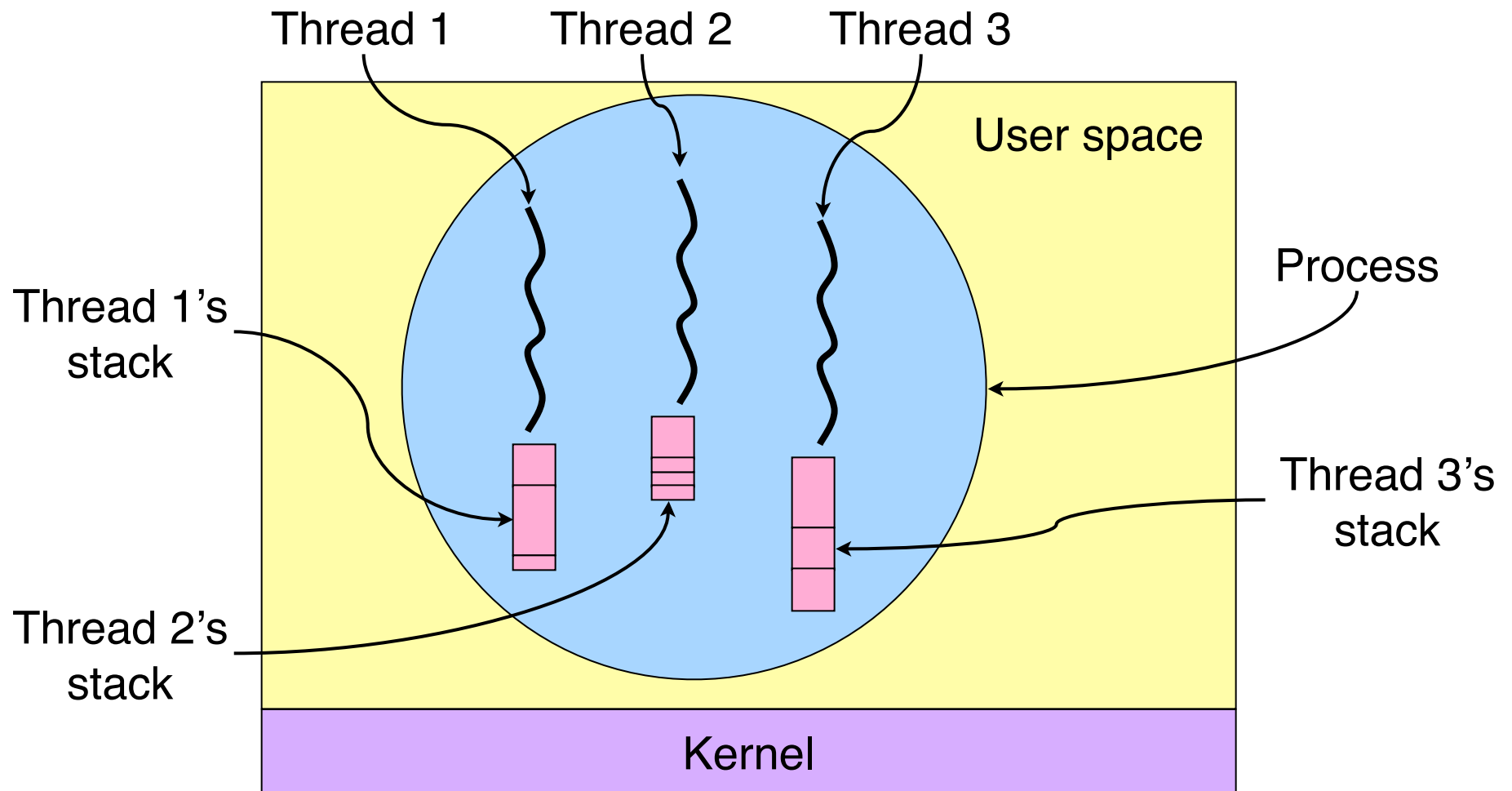
## Per thread items

Program counter  
Registers  
Stack & stack pointer  
State

## Per thread items

Program counter  
Registers  
Stack & stack pointer  
State

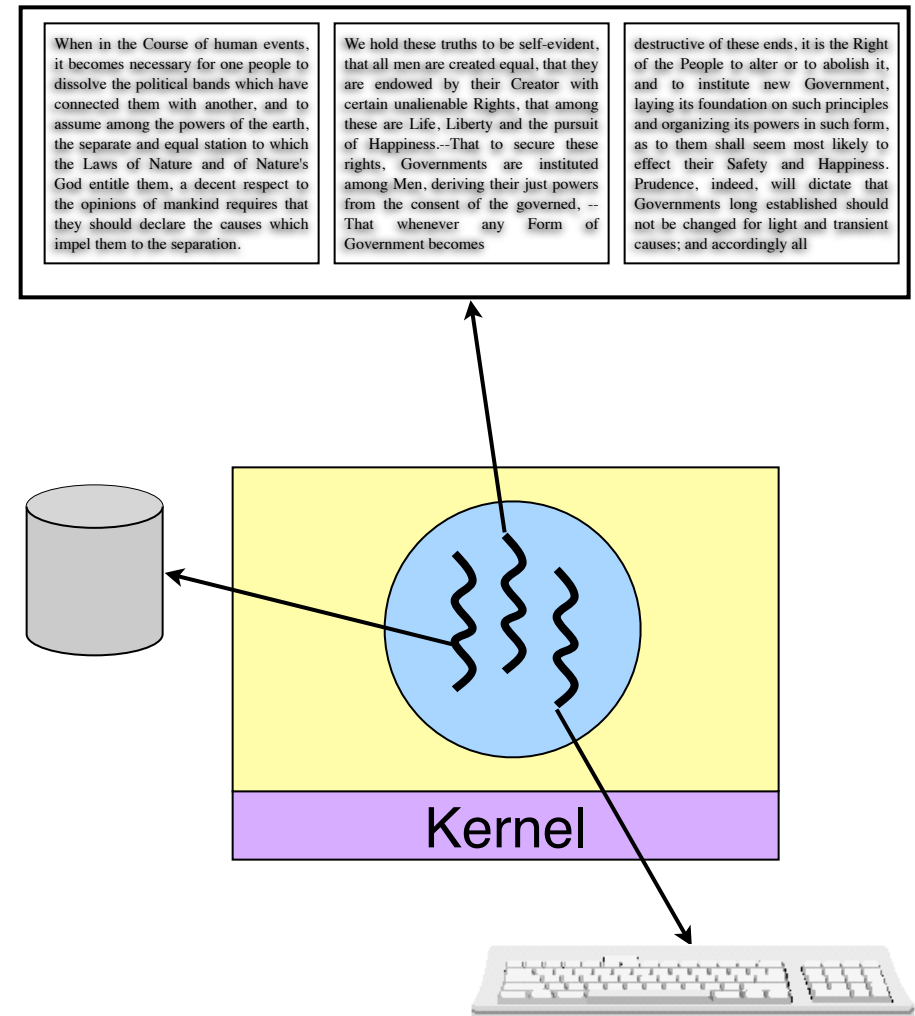
# Threads & stacks



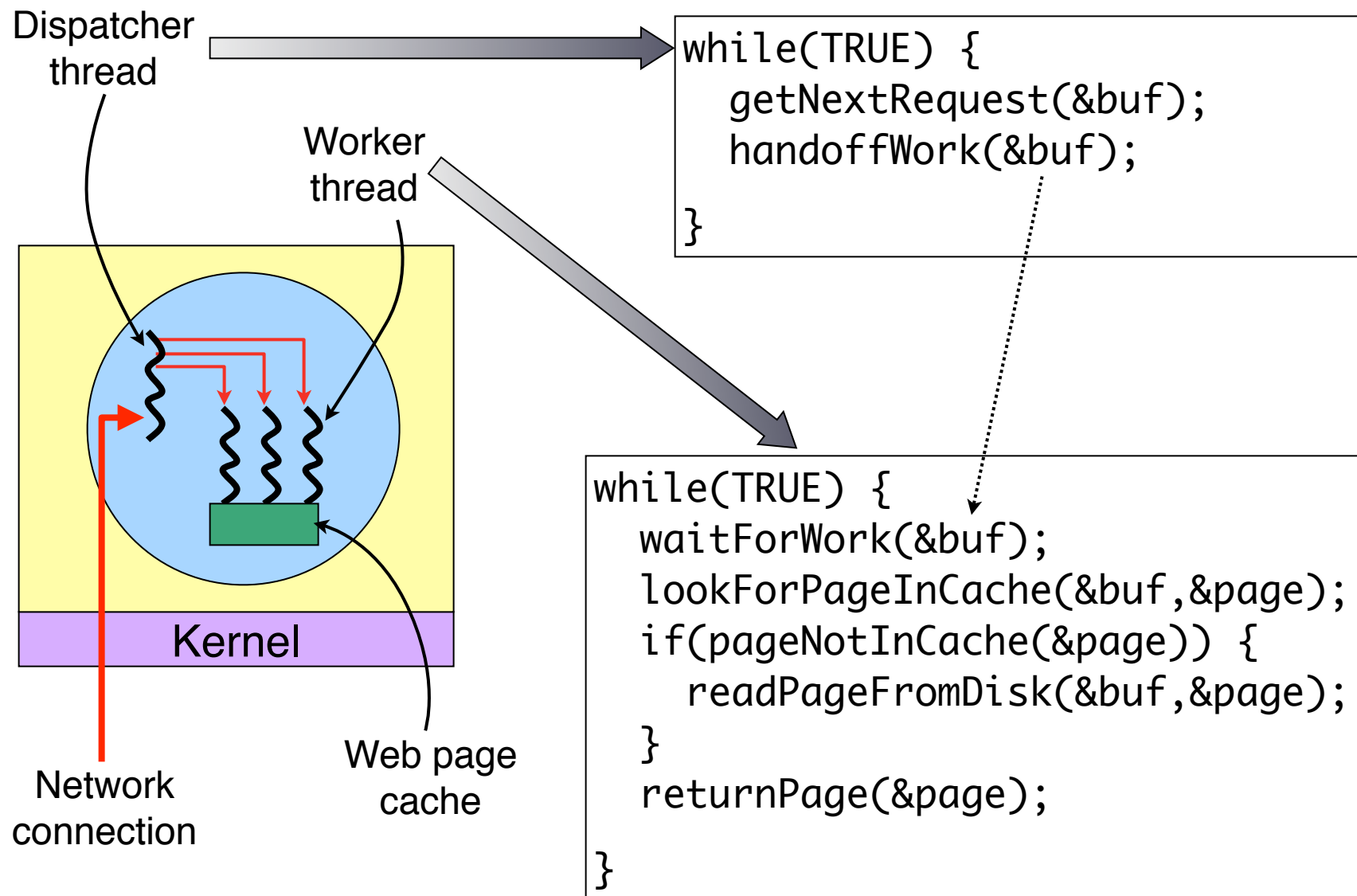
➡ Each thread has its own stack!

# Why use threads?

- ◆ Allow a single application to do many things at once
  - Simpler programming model
  - Less waiting
- ◆ Threads are faster to create or destroy
  - No separate address space
- ◆ Overlap computation and I/O
  - Could be done without threads, but it's harder
- ◆ Example: word processor
  - Thread to read from keyboard
  - Thread to format document
  - Thread to write to disk



# Multithreaded Web server



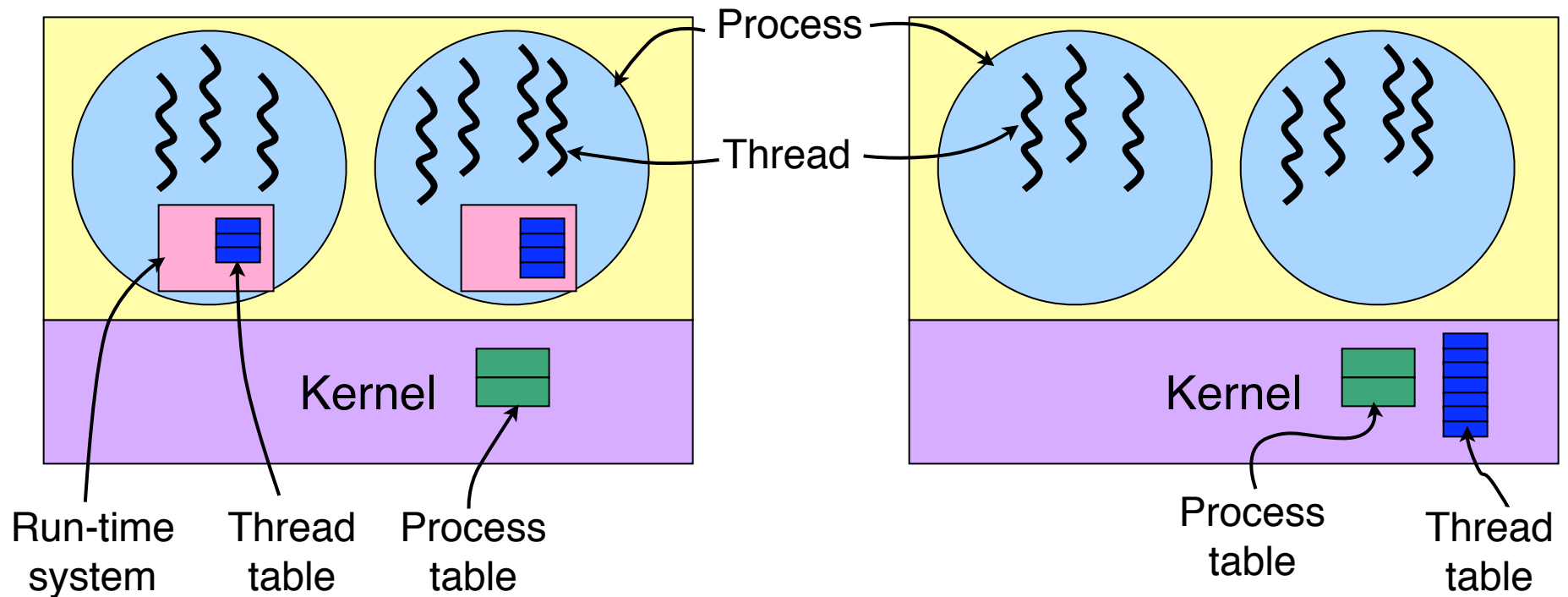


# Three ways to build a server

---

- ✦ Thread model
  - Parallelism
  - Blocking system calls
- ✦ Single-threaded process: slow, but easier to do
  - No parallelism
  - Blocking system calls
- ✦ Finite-state machine (event model)
  - Each activity has its own state
  - States change when system calls complete or interrupts occur
  - Parallelism
  - Nonblocking system calls
  - Interrupts

# Implementing threads



## User-level threads

- + No need for kernel support
- May be slower than kernel threads
- Harder to do non-blocking I/O

## Kernel-level threads

- + More flexible scheduling
- + Non-blocking I/O
- Not portable

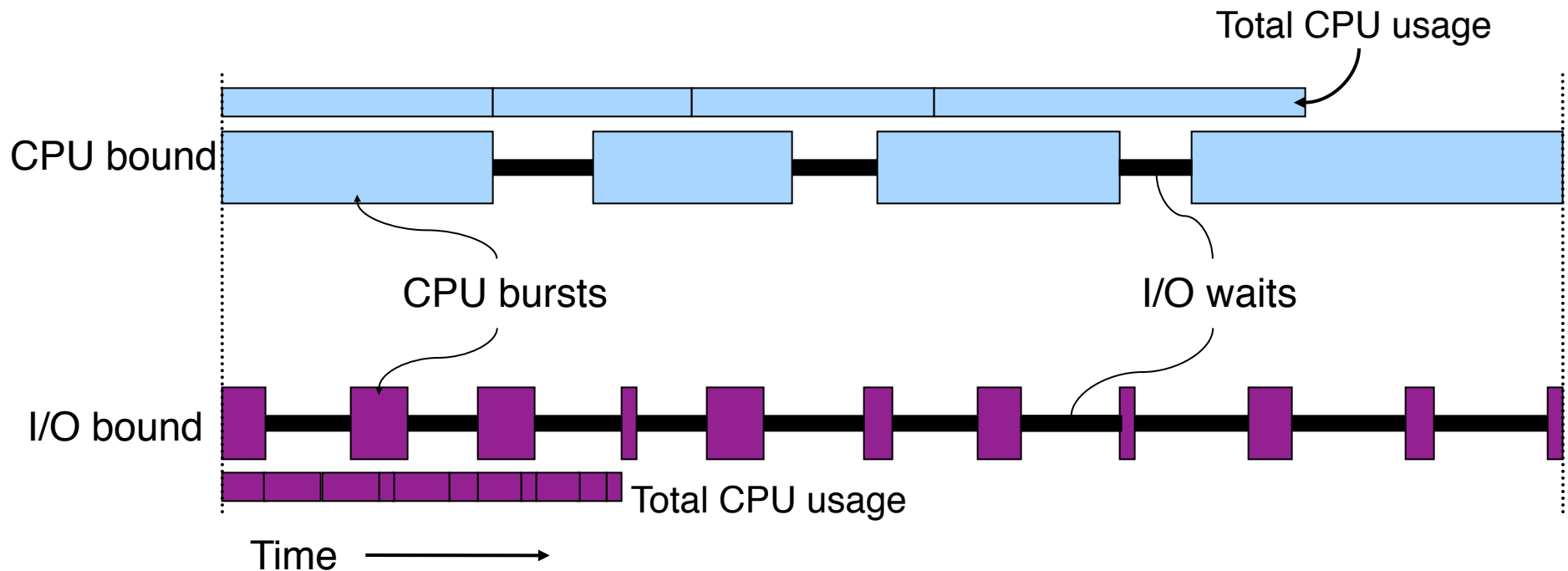
# Scheduling

---

- ✦ What is scheduling?
  - Goals
  - Mechanisms
- ✦ Scheduling on batch systems
- ✦ Scheduling on interactive systems
- ✦ Other kinds of scheduling
  - Real-time scheduling

# Why schedule processes?

- ✦ Bursts of CPU usage alternate with periods of I/O wait
- ✦ Some processes are **CPU-bound**: they don't make many I/O requests
- ✦ Other processes are **I/O-bound** and make many kernel requests



# When are processes scheduled?

---

- ✦ At the time they enter the system
  - Common in batch systems
  - Two types of batch scheduling
    - Submission of a new job causes the scheduler to run
    - Scheduling only done when a job voluntarily gives up the CPU (i.e., while waiting for an I/O request)
- ✦ At relatively fixed intervals (clock interrupts)
  - Necessary for interactive systems
  - May also be used for batch systems
  - Scheduling algorithms at each interrupt, and picks the next process from the pool of “ready” processes

# Scheduling goals

---

## ✦ All systems

- Fairness: give each process a fair share of the CPU
- Enforcement: ensure that the stated policy is carried out
- Balance: keep all parts of the system busy

## ✦ Batch systems

- Throughput: maximize jobs per unit time (hour)
- Turnaround time: minimize time users wait for jobs
- CPU utilization: keep the CPU as busy as possible

## ✦ Interactive systems

- Response time: respond quickly to users' requests
- Proportionality: meet users' expectations

## ✦ Real-time systems

- Meet deadlines: missing deadlines is a system failure!
- Predictability: same type of behavior for each time slice

# Measuring scheduling performance

---

- ✦ Throughput
  - Amount of work completed per second (minute, hour)
  - Higher throughput usually means better utilized system
- ✦ Response time
  - Response time is time from when a command is submitted until results are returned
  - Can measure average, variance, minimum, maximum, ...
  - May be more useful to measure time spent waiting
- ✦ Turnaround time
  - Like response time, but for batch jobs (response is the completion of the process)
- ✦ Usually not possible to optimize for **all** metrics with a single scheduling algorithm

# Interactive vs. batch scheduling

---

## Batch

First-Come-First-Served (FCFS)

Shortest Job First (SJF)

Shortest Remaining Time First (SRTF)

Priority (non-preemptive)

## Interactive

Round-Robin (RR)

Priority (preemptive)

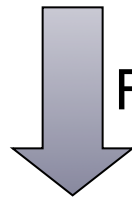
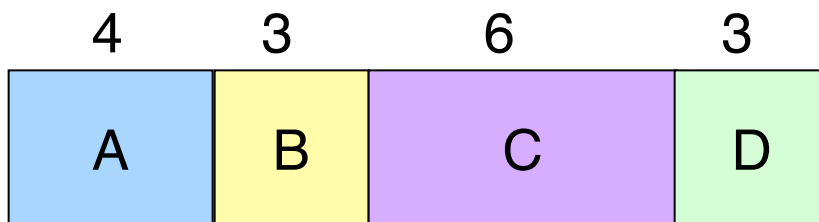
Multi-level feedback queue

Lottery scheduling



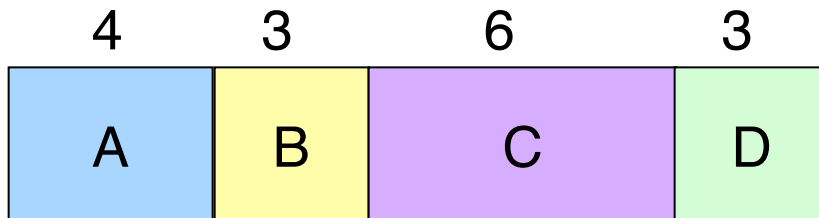
# First Come, First Served (FCFS)

Current job queue



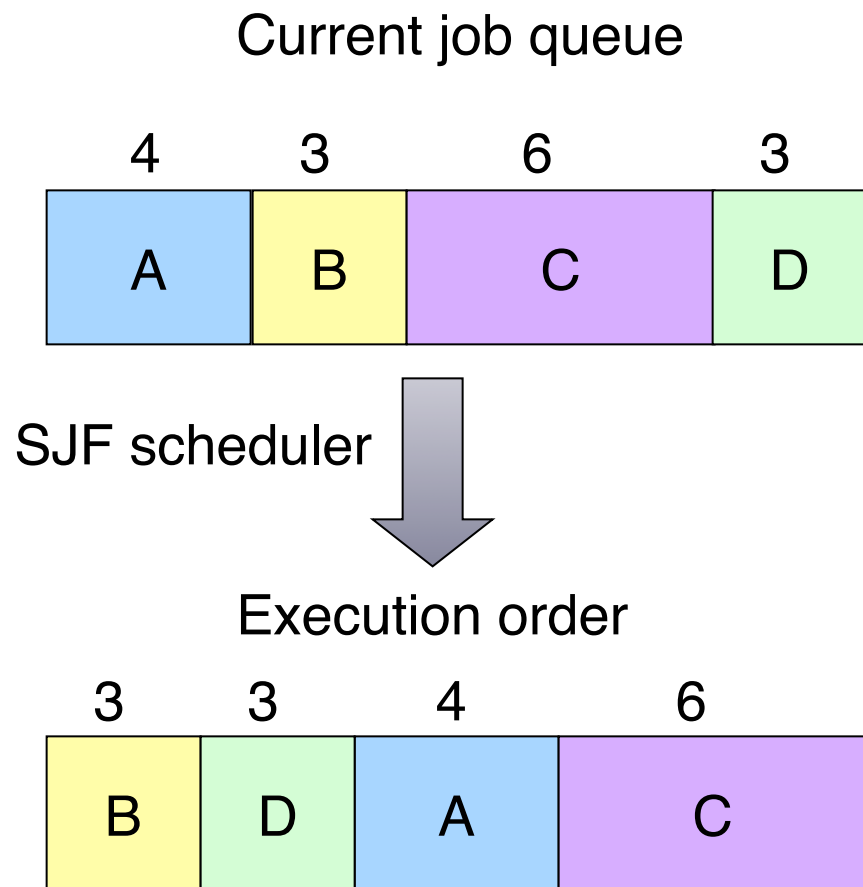
FCFS scheduler

Execution order



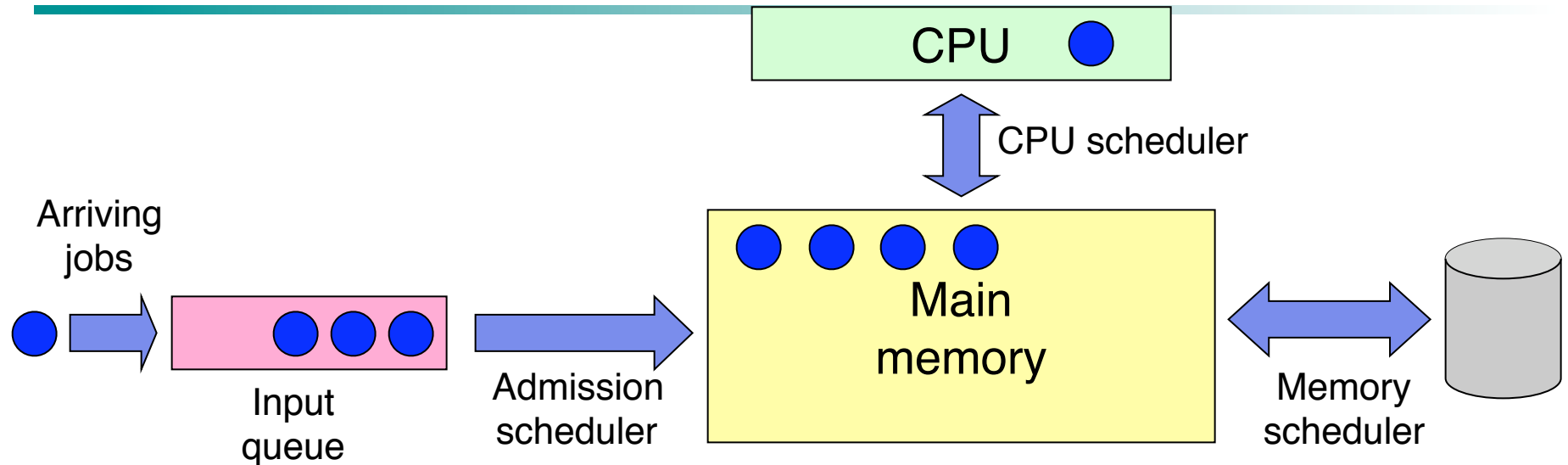
- ✦ Goal: do jobs in the order they arrive
  - Fair in the same way a bank teller line is fair
- ✦ Simple algorithm!
- ✦ Problem: long jobs delay every job after them
  - Many processes may wait for a single long job

# Shortest Job First (SJF)



- ♦ Goal: do the shortest job first
  - Short jobs complete first
  - Long jobs delay every job after them
- ♦ Jobs sorted in increasing order of execution time
  - Ordering of ties doesn't matter
- ♦ Shortest Remaining Time First (SRTF): preemptive form of SJF
  - Re-evaluate when a new job is submitted
- ♦ Problem: how does the scheduler know how long a job will take?

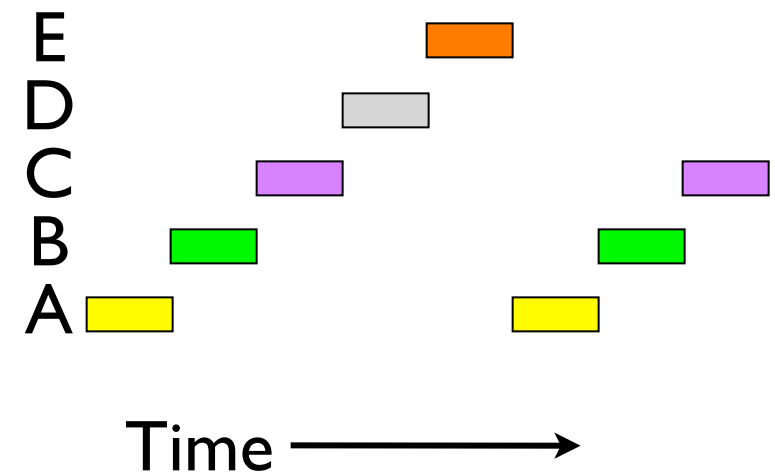
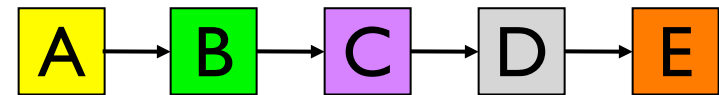
# Three-level scheduling



- ✦ Jobs held in input queue until moved into memory
  - Pick “complementary jobs”: small & large, CPU- & I/O-intensive
  - Jobs move into memory when admitted
- ✦ CPU scheduler picks next job to run
- ✦ Memory scheduler picks some jobs from main memory and moves them to disk if insufficient memory space

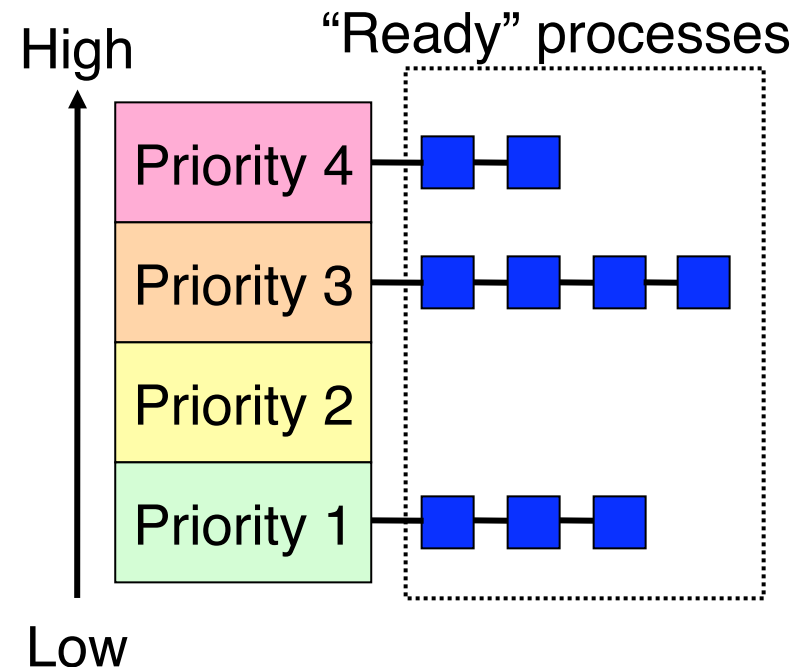
# Round Robin (RR) scheduling

- ◆ Round Robin scheduling
  - Give each process a **fixed time slot** (quantum)
  - Rotate through “ready” processes
  - Each process makes some progress
- ◆ What’s a good quantum?
  - Too short: many process switches hurt efficiency
  - Too long: poor response to interactive requests
  - Typical length: 10–100 ms



# Priority scheduling

- ◆ Assign a priority to each process
  - “Ready” process with highest priority allowed to run
  - Running process may be interrupted after its quantum expires
- ◆ Priorities may be assigned dynamically
  - Reduced when a process uses CPU time
  - Increased when a process waits for I/O
- ◆ Often, processes grouped into multiple queues based on priority, and run round-robin per queue



# Shortest process next

---

- ✦ Run the process that will finish the soonest
  - In interactive systems, job completion time is unknown!
- ✦ Guess at completion time based on previous runs
  - Update estimate each time the job is run
  - Estimate is a combination of previous estimate and most recent run time
- ✦ Not often used because round robin with priority works so well!

# Lottery scheduling

---

- ✦ Give processes “tickets” for CPU time
  - More tickets => higher share of CPU
- ✦ Each quantum, pick a ticket at random
  - If there are  $n$  tickets, pick a number from 1 to  $n$
  - Process holding the ticket gets to run for a quantum
- ✦ Over the long run, each process gets the CPU  $m/n$  of the time if the process has  $m$  of the  $n$  existing tickets
- ✦ Tickets can be transferred
  - Cooperating processes can exchange tickets
  - Clients can transfer tickets to server so it can have a higher priority

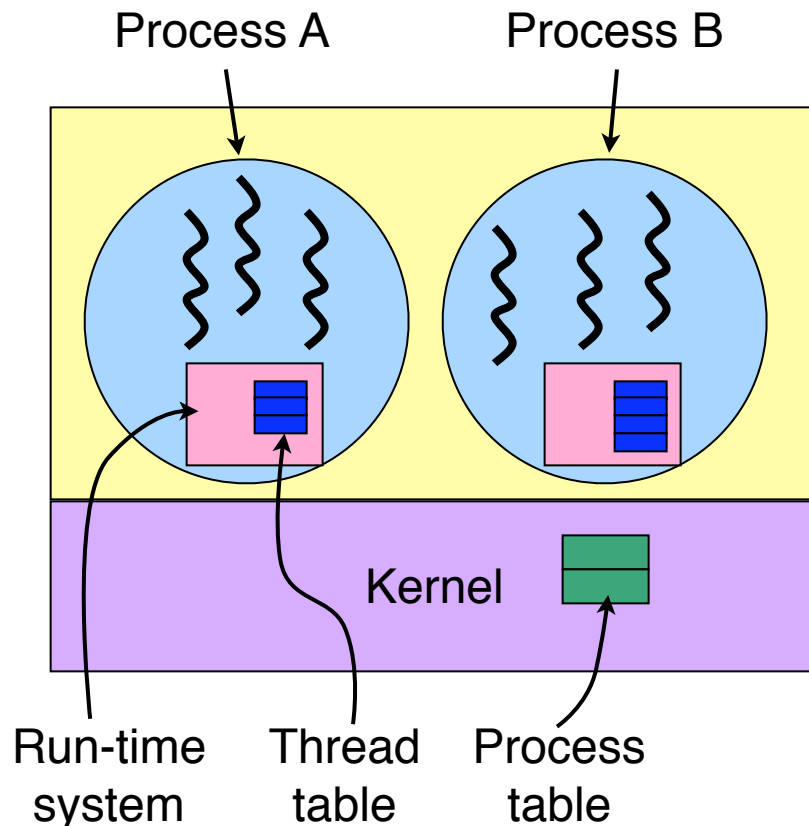
# Policy versus mechanism

---

- ✦ Separate what may be done from how it is done
  - Mechanism allows
    - Priorities to be assigned to processes
    - CPU to select processes with high priorities
  - Policy set by what priorities are assigned to processes
- ✦ Scheduling algorithm parameterized
  - Mechanism in the kernel
  - Priorities assigned in the kernel or by users
- ✦ Parameters may be set by user processes
  - Don't allow a user process to take over the system!
  - Allow a user process to voluntarily lower its own priority
  - Allow a user process to assign priority to its threads

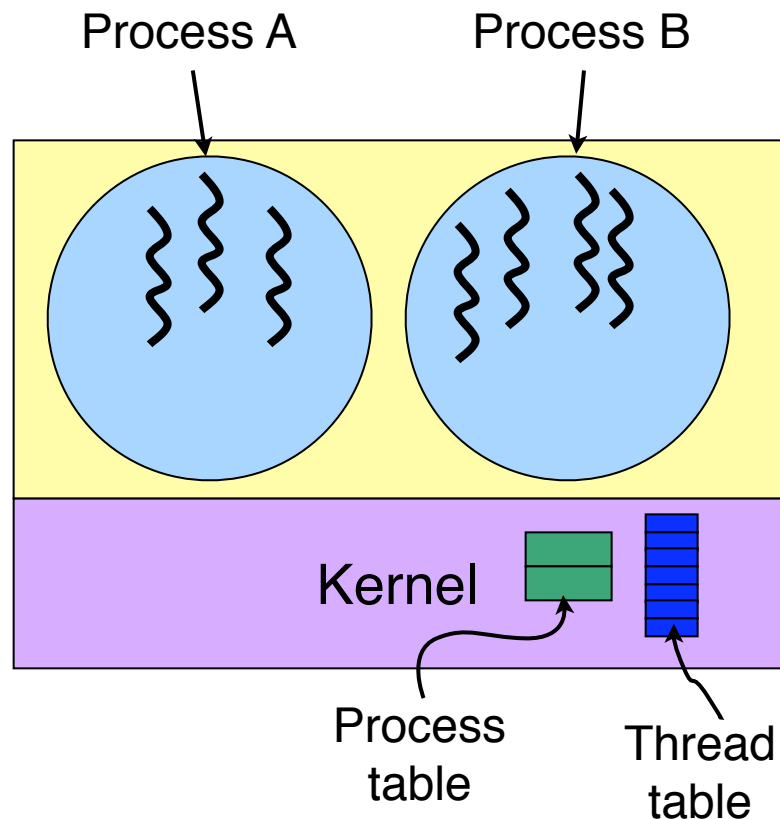


# Scheduling user-level threads



- ✦ Kernel picks a process to run next
- ✦ Run-time system (at user level) schedules threads
  - Run each thread for less than process quantum
  - Example: processes get 40ms each, threads get 10ms each
- ✦ Example schedule:  
A1,A2,A3,A1,B1,B3,B2,B3
- ✦ Not possible:  
A1,A2,B1,B2,A3,B3,A2,B1

# Scheduling kernel-level threads



- ✦ Kernel schedules each thread
  - No restrictions on ordering
  - May be more difficult for each process to specify priorities
- ✦ Example schedule:  
A1,A2,A3,A1,  
B1,B3,B2,B3
- ✦ Also possible:  
A1,A2,B1,B2,  
A3,B3,A2,B1

# **Chapter 2: Processes & Threads**

Part 2:  
Interprocess Communication & Synchronization

# Why do we need IPC?

---

- ✦ Each process operates sequentially
- ✦ All is fine until processes want to share data
  - Exchange data between multiple processes
  - Allow processes to navigate critical regions
  - Maintain proper sequencing of actions in multiple processes
- ✦ These issues apply to threads as well
  - Threads can share data easily (same address space)
  - Other two issues apply to threads

# Example: bounded buffer problem

## Shared variables

```
const int n;  
typedef ... Item;  
Item buffer[n];  
int in = 0, out = 0,  
    counter = 0;
```

## Producer

```
Item pitm;  
while (1) {  
    ...  
    produce an item into pitm  
    ...  
    while (counter == n)  
        ;  
    buffer[in] = pitm;  
    in = (in+1) % n;  
    counter += 1;  
}
```

## Atomic statements:

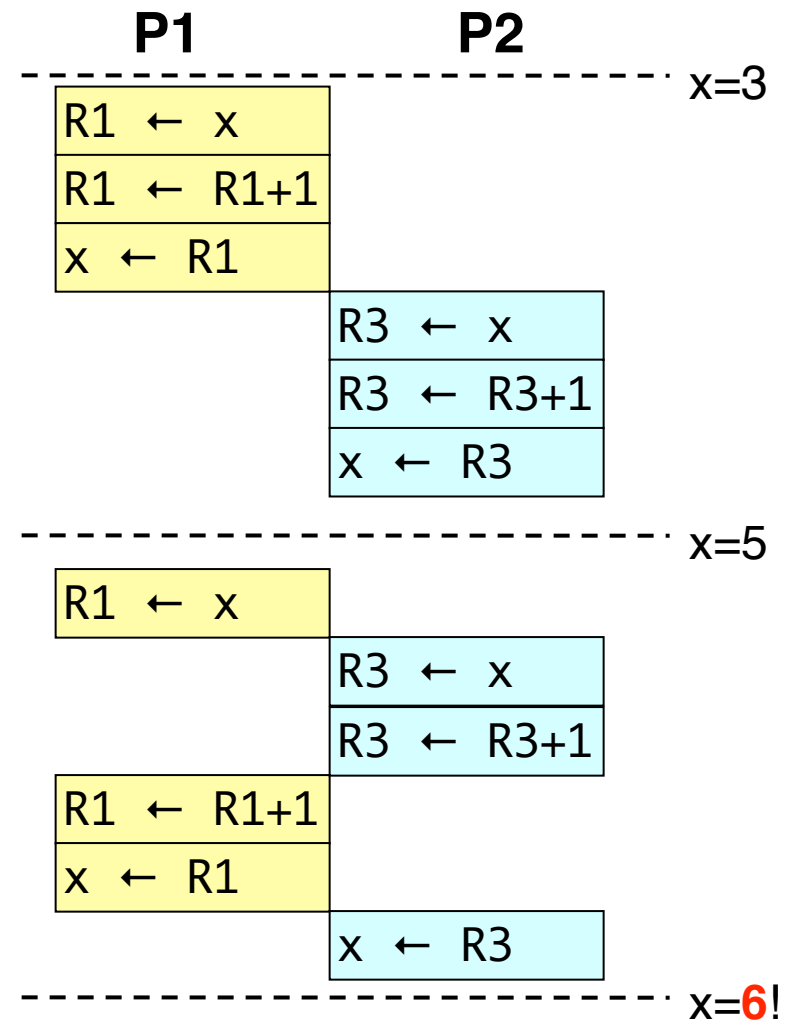
```
Counter += 1;  
Counter -= 1;
```

## Consumer

```
Item citm;  
while (1) {  
    while (counter == 0)  
        ;  
    citm = buffer[out];  
    out = (out+1) % n;  
    counter -= 1;  
    ...  
    consume the item in citm  
    ...  
}
```

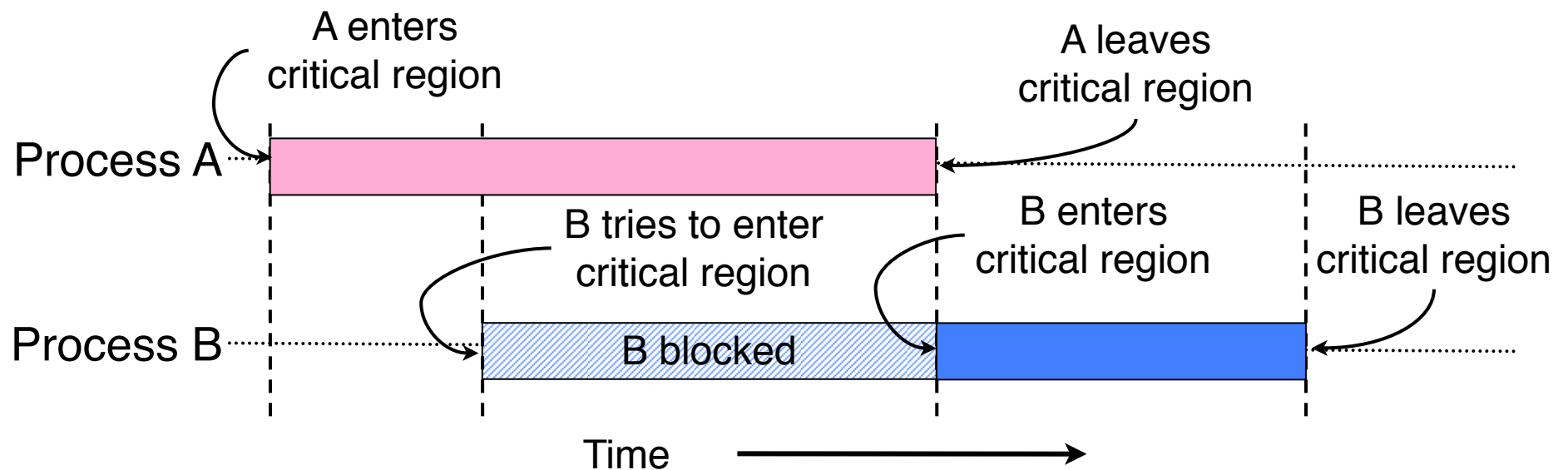
# Problem: race conditions

- ✦ Cooperating processes share storage (memory)
- ✦ Both may read and write the shared memory
- ✦ Problem: can't guarantee that read followed by write is atomic
  - Ordering matters!
- ✦ This can result in erroneous results!
- ✦ We need to eliminate race conditions...



# Critical regions

- ◆ Use critical regions to provide mutual exclusion and help fix race conditions
- ◆ Four conditions to provide mutual exclusion
  - No two processes simultaneously in critical region
  - No assumptions made about speeds or number of CPUs
  - No process running outside its critical region may block another process
  - A process may not wait forever to enter its critical region



# Busy waiting: strict alternation

## Process 0

```
while (TRUE) {  
    while (turn != 0)  
        ; /* loop */  
    critical_region ();  
    turn = 1;  
    noncritical_region ();  
}
```

## Process 1

```
while (TRUE) {  
    while (turn != 1)  
        ; /* loop */  
    critical_region ();  
    turn = 0;  
    noncritical_region ();  
}
```

- ✦ Use a shared variable (turn) to keep track of whose turn it is
- ✦ Waiting process continually reads the variable to see if it can proceed
  - This is called a spin lock because the waiting process “spins” in a tight loop reading the variable
- ✦ Avoids race conditions, but doesn't satisfy criterion 3 for critical regions



# Busy waiting: working solution

```
#define FALSE 0
#define TRUE 1
#define N 2 // # of processes
int turn; // Whose turn is it?
int interested[N]; // Set to 1 if process j is interested

void enter_region(int process)
{
    int other = 1-process; // # of the other process
    interested[process] = TRUE; // show interest
    turn = process; // Set it to my turn
    while (turn==process && interested[other]==TRUE)
        ; // Wait while the other process runs
}

void leave_region (int process)
{
    interested[process] = FALSE; // I'm no longer interested
}
```

# Bakery algorithm for many processes

## ◆ Notation used

- $\lll$  is lexicographical order on (ticket#, process ID)
- $(a,b) \lll (c,d)$  if  $(a \lll c)$  or  $((a==c) \text{ and } (b<d))$
- $\text{Max}(a_0, a_1, \dots, a_{n-1})$  is a number  $k$  such that  $k \geq a_i$  for all  $i$

## ◆ Shared data

- choosing initialized to 0
- number initialized to 0

```
int n; // # of processes
int choosing[n];
int number[n];
```

# Bakery algorithm: code

```
while (1) { // i is the number of the current process
    choosing[i] = 1;
    number[i] = max(number[0], number[1], ..., number[n-1]) + 1;
    choosing[i] = 0;
    for (j = 0; j < n; j++) {
        while (choosing[j]) // wait while j is choosing a
            ; // number
        // Wait while j wants to enter and j <<< i
        while ((number[j] != 0) &&
            ((number[j] < number[i]) ||
            (number[j] == number[i]) && (j < i))))
            ;
    }
    // critical section
    number[i] = 0;
    // rest of code
}
```

# Hardware for synchronization

---

- ✦ Prior methods work, but...
  - May be somewhat complex
  - Require busy waiting: process spins in a loop waiting for something to happen, wasting CPU time
- ✦ Solution: use hardware
- ✦ Several hardware methods
  - Test & set: test a variable and set it in one instruction
  - Atomic swap: switch register & memory in one instruction
  - Turn off interrupts: process won't be switched out unless it asks to be suspended

# Mutual exclusion using hardware

- ✦ Single shared variable lock
- ✦ Still requires busy waiting, but code is much simpler
- ✦ Two versions
  - Test and set
  - Swap
- ✦ Works for any number of processes
- ✦ Possible problem with requirements
  - Non-concurrent code can lead to unbounded waiting

```
int lock = 0;
```

## Code for process $P_i$

```
while (1) {  
    while (TestAndSet(lock))  
        ;  
    // critical section  
    lock = 0;  
    // remainder of code  
}
```

## Code for process $P_i$

```
while (1) {  
    while (Swap(lock,1) == 1)  
        ;  
    // critical section  
    lock = 0;  
    // remainder of code  
}
```

# Eliminating busy waiting

- ◆ Problem: previous solutions waste CPU time
  - Both hardware and software solutions require spin locks
  - Allow processes to sleep while they wait to execute their critical sections
- ◆ Problem: priority inversion (higher priority process waits for lower priority process)
- ◆ Solution: use semaphores
  - Synchronization mechanism that doesn't require busy waiting during entire critical section
- ◆ Implementation
  - Semaphore  $S$  accessed by two **atomic** operations
    - $\text{Down}(S)$ : while ( $S \leq 0$ ) {};  $S = S - 1$ ;
    - $\text{Up}(S)$ :  $S = S + 1$ ;
  - $\text{Down}()$  is another name for  $P()$
  - $\text{Up}()$  is another name for  $V()$
  - Modify implementation to eliminate busy wait from  $\text{Down}()$

# Critical sections using semaphores

- ✦ Define a class called Semaphore
  - Class allows more complex implementations for semaphores
  - Details hidden from processes
- ✦ Code for individual process is simple

## Shared variables

```
Semaphore mutex;
```

## Code for process $P_i$

```
while (1) {  
    down(mutex);  
    // critical section  
    up(mutex);  
    // remainder of code  
}
```

# Implementing semaphores with blocking

- ◆ Assume two operations:
  - Sleep(): suspends current process
  - Wakeup(P): allows process P to resume execution
- ◆ Semaphore is a class
  - Track value of semaphore
  - Keep a list of processes waiting for the semaphore
- ◆ Operations still atomic

```
class Semaphore {  
    int value;  
    ProcessList pl;  
    void down ();  
    void up ();  
};
```

## Semaphore code

```
Semaphore::down ()  
{  
    value -= 1;  
    if (value < 0) {  
        // add this process to pl  
        Sleep ();  
    }  
}  
  
Semaphore::up () {  
    Process P;  
    value += 1;  
    if (value <= 0) {  
        // remove a process P  
        // from pl  
        Wakeup (P);  
    }  
}
```



# Semaphores for general synchronization

- ✦ We want to execute B in P1 only after A executes in P0
- ✦ Use a semaphore initialized to 0
- ✦ Use up() to notify P1 at the appropriate time

## Shared variables

```
// flag initialized to 0  
Semaphore flag;
```

### Process P<sub>0</sub>

```
.  
.   
.   
// Execute code for A  
flag.up ();
```

### Process P<sub>1</sub>

```
.  
.   
.   
flag.down ();  
// Execute code for B
```

# Types of semaphores

---

- ✦ Two different types of semaphores
  - Counting semaphores
  - Binary semaphores
- ✦ Counting semaphore
  - Value can range over an unrestricted range
- ✦ Binary semaphore
  - Only two values possible
    - 1 means the semaphore is available
    - 0 means a process has acquired the semaphore
  - May be simpler to implement
- ✦ Possible to implement one type using the other

# Monitors

- ✦ A monitor is another kind of **high-level** synchronization primitive
  - One monitor has multiple entry points
  - Only one process may be in the monitor at any time
  - Enforces mutual exclusion - less chance for programming errors
- ✦ Monitors provided by high-level language
  - Variables belonging to monitor are protected from simultaneous access
  - Procedures in monitor are guaranteed to have mutual exclusion
- ✦ Monitor implementation
  - Language / compiler handles implementation
  - Can be implemented using semaphores

# Monitor usage

- ✦ This looks like C++ code, but it's not supported by C++
- ✦ Provides the following features:
  - Variables foo, bar, and arr are accessible only by proc1 & proc2
  - **Only one process** can be executing in either proc1 or proc2 at any time

```
monitor mon {  
    int foo;  
    int bar;  
    double arr[100];  
    void proc1(...) {  
    }  
    void proc2(...) {  
    }  
    void mon() { // initialization code  
    }  
}
```

# Condition variables in monitors

- ✦ Problem: how can a process wait inside a monitor?
  - Can't simply sleep: there's no way for anyone else to enter
  - Solution: use a **condition variable**
- ✦ Condition variables support two operations
  - Wait(): suspend this process until signaled
  - Signal(): wake up exactly one process waiting on this condition variable
    - If no process is waiting, signal has no effect
    - Signals on condition variables aren't "saved up"
- ✦ Condition variables are only usable within monitors
  - Process must be in monitor to signal on a condition variable
  - Question: which process gets the monitor after Signal()?

# Monitor semantics

- ✦ Problem: P signals on condition variable X, waking Q
  - Both can't be active in the monitor at the same time
  - Which one continues first?
- ✦ **Mesa** semantics
  - Signaling process (P) continues first
  - Q resumes when P leaves the monitor
  - Seems more logical: why suspend P when it signals?
- ✦ **Hoare** semantics
  - Awakened process (Q) continues first
  - P resumes when Q leaves the monitor
  - May be better: condition that Q wanted may no longer hold when P leaves the monitor

# Locks & condition variables

- ✦ Monitors require native language support
- ✦ Provide monitor support using special data types and procedures
  - Locks (Acquire(), Release())
  - Condition variables (Wait(), Signal())
- ✦ Lock usage
  - Acquiring a lock == entering a monitor
  - Releasing a lock == leaving a monitor
- ✦ Condition variable usage
  - Each condition variable is associated with exactly one lock
  - Lock must be held to use condition variable
  - Waiting on a condition variable releases the lock implicitly
  - Returning from Wait() on a condition variable reacquires the lock

# Implementing locks with semaphores

```
class Lock {  
    Semaphore mutex(1);  
    Semaphore next(0);  
    int nextCount = 0;  
};
```

```
Lock::Acquire()  
{  
    mutex.down();  
}
```

```
Lock::Release()  
{  
    if (nextCount > 0)  
        next.up();  
    else  
        mutex.up();  
}
```

- ✦ Use *mutex* to ensure **exclusion** within the lock bounds
- ✦ Use *next* to give lock to processes with a higher priority (why?)
- ✦ *nextCount* indicates whether there are any higher priority waiters



# Implementing condition variables

```
class Condition {  
    Lock *lock;  
    Semaphore condSem(0);  
    int semCount = 0;  
};
```

```
Condition::Wait ()  
{  
    semCount += 1;  
    if (lock->nextCount > 0)  
        lock->next.up();  
    else  
        lock->mutex.up();  
    condSem.down ();  
    semCount -= 1;  
}
```

✦ Are these Hoare or **Mesa** semantics?

✦ Can there be multiple condition variables for a single Lock?

```
Condition::Signal ()  
{  
    if (semCount > 0) {  
        lock->nextCount += 1;  
        condSem.up ();  
        lock->next.down ();  
        lock->nextCount -= 1;  
    }  
}
```

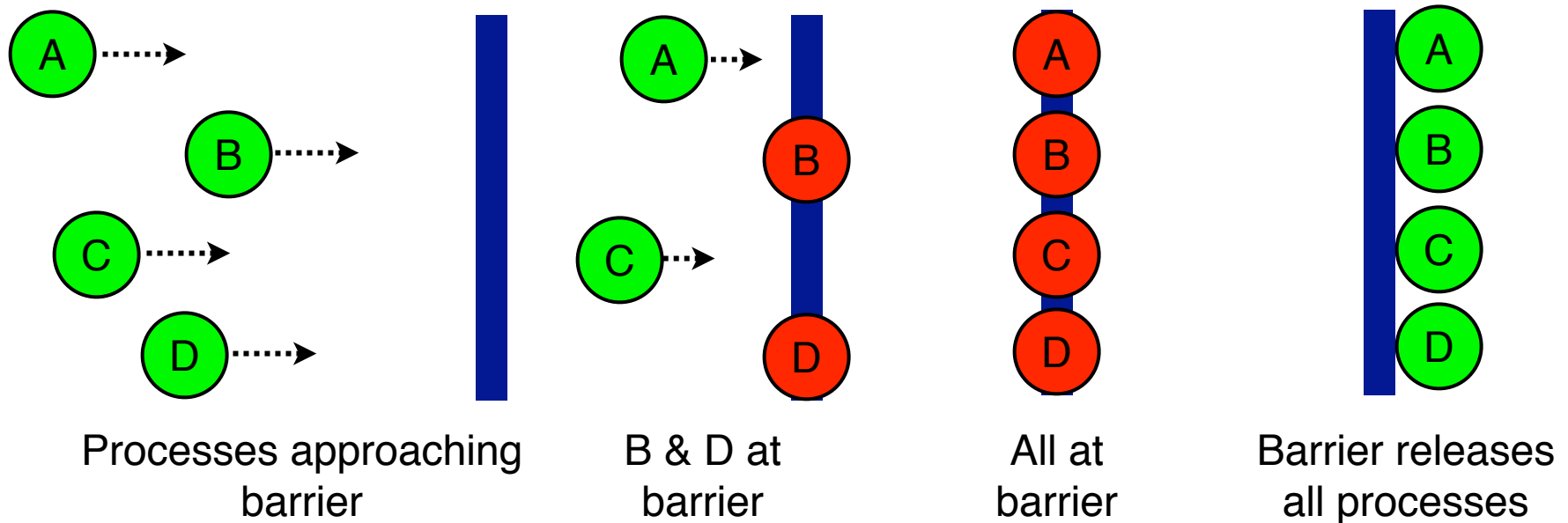
# Message passing

---

- ✦ Synchronize by exchanging messages
- ✦ Two primitives:
  - Send: send a message
  - Receive: receive a message
  - Both may specify a “channel” to use
- ✦ Issue: how does the sender know the receiver got the message?
- ✦ Issue: authentication

# Barriers

- ✦ Used for **synchronizing** multiple processes
- ✦ Processes wait at a “barrier” until all in the group arrive
- ✦ After all have arrived, all processes can proceed
- ✦ May be implemented using locks and condition variables



# Implementing barriers using semaphores

```
Barrier b;          /* contains two semaphores */
b.bsem.value = 0;   /* for the barrier */
b.mutex.value = 1;  /* for mutual exclusion */
b.waiting = 0;
b.maxproc = n;      /* n processes needed at barrier */
```

```
HitBarrier (Barrier *b)
{
    SemDown (&b->mutex);
    if (++b->waiting >= b->maxproc) {
        while (--b->waiting > 0) {
            SemUp (&b->bsem);
        }
        SemUp (&b->mutex);
    } else {
        SemUp (&b->mutex);
        SemDown (&b->bsem);
    }
}
```

Use locks and  
condition variables

# Deadlock and starvation

- ◆ Deadlock: two or more processes are **waiting** indefinitely for an event that can only be caused by a waiting process
  - P<sub>0</sub> gets A, needs B
  - P<sub>1</sub> gets B, needs A
  - Each process waiting for the other to signal
- ◆ Starvation: indefinite blocking
  - Process is never removed from the semaphore queue in which it is suspended
  - May be caused by ordering in queues (priority)

## Shared variables

```
Semaphore A(1), B(1);
```

### Process P<sub>0</sub>

```
A.down();  
B.down();  
.  
.  
.  
B.up();  
A.up();
```

### Process P<sub>1</sub>

```
B.down();  
A.down();  
.  
.  
.  
A.up();  
B.up();
```

# Classical synchronization problems

---

- ✦ Bounded Buffer
  - Multiple producers and consumers
  - Synchronize access to shared buffer
- ✦ Readers & Writers
  - Many processes that may read and/or write
  - Only one writer allowed at any time
  - Many readers allowed, but not while a process is writing
- ✦ Dining Philosophers
  - Resource allocation problem
  - $N$  processes and limited resources to perform sequence of tasks
- ✦ Goal: use **semaphores** to implement solutions to these problems

# Bounded buffer problem

Goal: implement producer-consumer without busy waiting

```
const int n;  
Semaphore empty(n), full(0), mutex(1);  
Item buffer[n];
```

## Producer

```
int in = 0;  
Item pitem;  
while (1) {  
    // produce an item  
    // into pitem  
    empty.down();  
    mutex.down();  
    buffer[in] = pitem;  
    in = (in+1) % n;  
    mutex.up();  
    full.up();  
}
```

## Consumer

```
int out = 0;  
Item citem;  
while (1) {  
    full.down();  
    mutex.down();  
    citem = buffer[out];  
    out = (out+1) % n;  
    mutex.up();  
    empty.up();  
    // consume item from  
    // citem  
}
```

# Readers-writers problem

## Shared variables

```
int nreaders;  
Semaphore mutex(1), writing(1);
```

## Reader process

```
...  
mutex.down();  
nreaders += 1;  
if (nreaders == 1) // wait if  
    writing.down(); // 1st reader  
mutex.up();  
// Read some stuff  
mutex.down();  
nreaders -= 1;  
if (nreaders == 0) // signal if  
    writing.up();    // last reader  
mutex.up();  
...
```

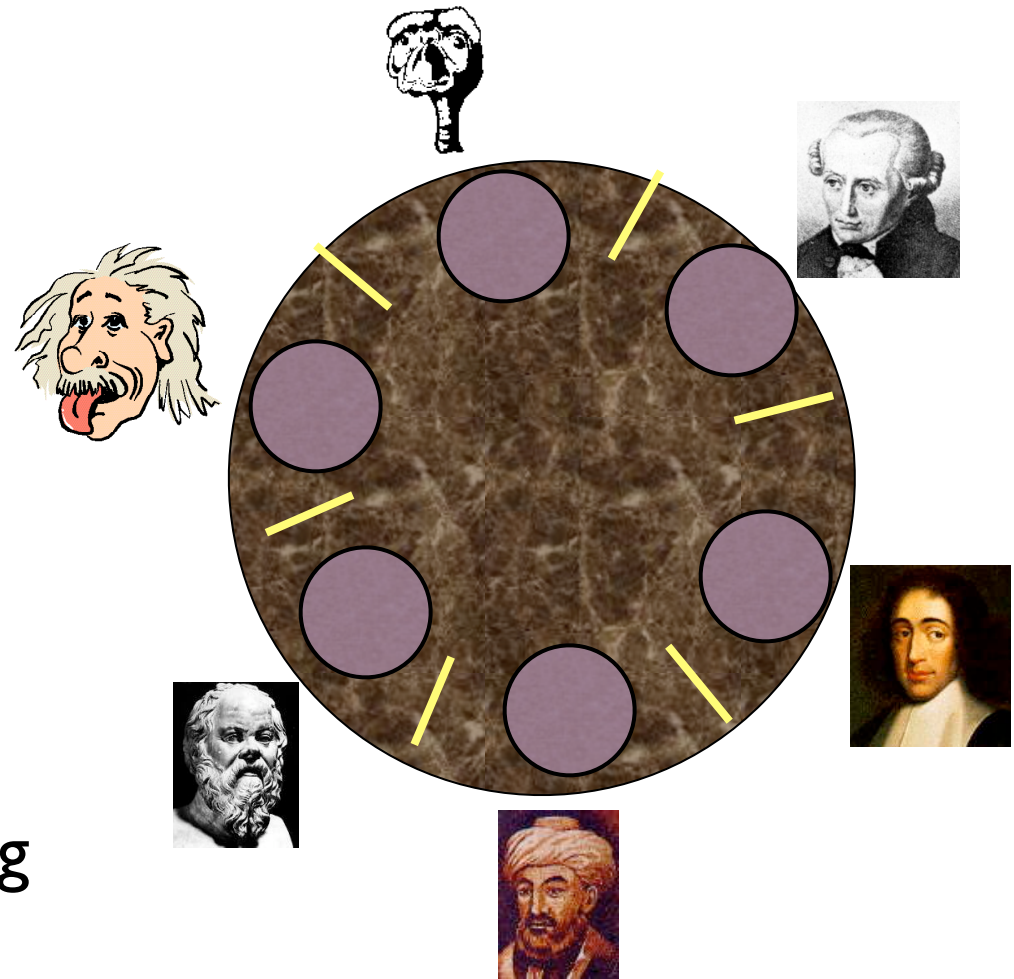
## Writer process

```
...  
writing.down();  
// Write some stuff  
writing.up();  
...
```



# Dining Philosophers

- ♦  $N$  philosophers around a table
  - All are hungry
  - All like to think
- ♦  $N$  chopsticks available
  - 1 between each pair of philosophers
- ♦ Philosophers need two chopsticks to eat
- ♦ Philosophers alternate between eating and thinking
- ♦ Goal: **coordinate use** of chopsticks



# Dining Philosophers: solution

## 1

- ✦ Use a semaphore for each chopstick
- ✦ A hungry philosopher
  - Gets the chopstick to his right
  - Gets the chopstick to his left
  - Eats
  - Puts down the chopsticks
- ✦ Potential problems?
  - Deadlock
  - Fairness

### Shared variables

```
const int n;  
// initialize to 1  
Semaphore chopstick[n];
```

### Code for philosopher *i*

```
while(1) {  
    chopstick[i].down();  
    chopstick[(i+1)%n].down();  
    // eat  
    chopstick[i].up();  
    chopstick[(i+1)%n].up();  
    // think  
}
```

# Dining Philosophers: solution

## 2

- ✦ Use a semaphore for each chopstick
- ✦ A hungry philosopher
  - Gets lower, then higher numbered chopstick
  - Eats
  - Puts down the chopsticks
- ✦ Potential problems?
  - Deadlock
  - Fairness

### Shared variables

```
const int n;  
// initialize to 1  
Semaphore chopstick[n];
```

### Code for philosopher *i*

```
int i1,i2;  
while(1) {  
    if (i != (n-1)) {  
        i1 = i;  
        i2 = i+1;  
    } else {  
        i1 = 0;  
        i2 = n-1;  
    }  
    chopstick[i1].down();  
    chopstick[i2].down();  
    // eat  
    chopstick[i1].up();  
    chopstick[i2].up();  
    // think  
}
```

# Dining philosophers with locks

## Shared variables

```
const int n;  
// initialize to THINK  
int state[n];  
Lock mutex;  
// use mutex for self  
Condition self[n];
```

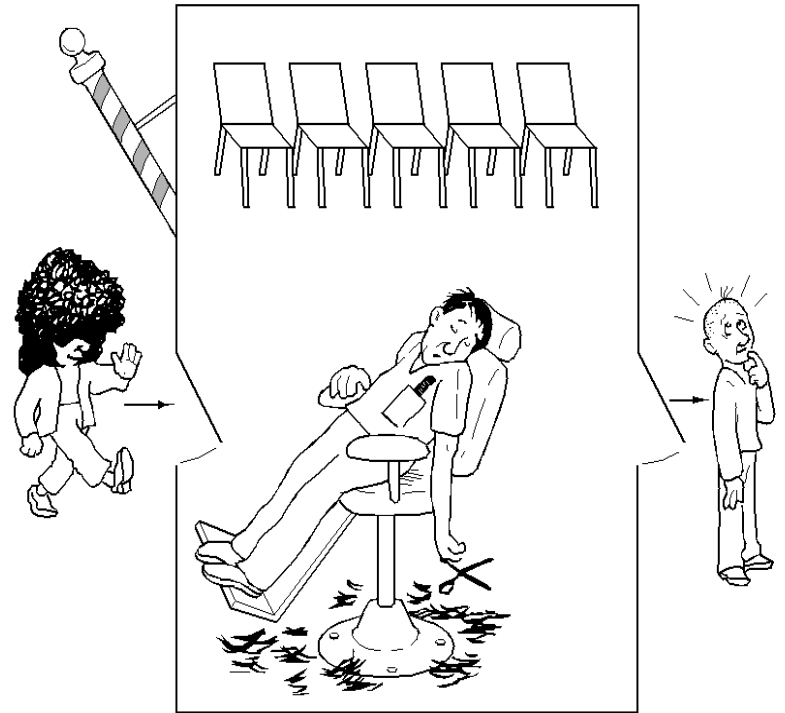
```
void test(int k)  
{  
    if ((state[(k+n-1)%n])!=EAT)  
&&  
        (state[k]==HUNGRY) &&  
        (state[(k+1)%n]!=EAT)) {  
        state[k] = EAT;  
        self[k].Signal();  
    }  
}
```

## Code for philosopher *j*

```
while (1) {  
    // pickup chopstick  
    mutex.Acquire();  
    state[j] = HUNGRY;  
    test(j);  
    if (state[j] != EAT)  
        self[j].Wait();  
    mutex.Release();  
    // eat  
    mutex.Acquire();  
    state[j] = THINK;  
    test((j+1)%n); // next  
    test((j+n-1)%n); // prev  
    mutex.Release();  
    // think  
}
```

# The Sleepy Barber Problem

- ♦ Barber wants to sleep all day
  - Wakes up to cut hair
- ♦ Customers wait in chairs until barber chair is free
  - Limited space in the waiting room
  - Leave if no space free
- ♦ Write the synchronization code for this problem...



# Code for the Sleepy Barber Problem

```
#define CHAIRS 5
Semaphore customers=0;
Semaphore barbers=0;
Semaphore mutex=0;
int waiting=0;
```

```
void barber(void)
{
    while(TRUE) {
        // Sleep if no customers
        customers.down();
        // Decrement # of waiting people
        mutex.down();
        waiting -= 1;
        // Wake up a customer to cut hair
        barbers.up();
        mutex.up();
        // Do the haircut
        cut_hair();
    }
}
```

```
void customer(void)
{
    mutex.down();
    // If there is space in the chairs
    if (waiting<CHAIRS) {
        // Another customer is waiting
        waiting++;
        // Wake up the barber. This is
        // saved up, so the barber doesn't
        // sleep if a customer is waiting
        customers.up();
        mutex.up();
        // Sleep until the barber is ready
        barbers.down();
        get_haircut();
    } else {
        // Chairs full, leave the critical
        // region
        mutex.up ();
    }
}
```