Chapter 2: Processes & Threads

Part 1: Processes & Scheduling

Processes and threads

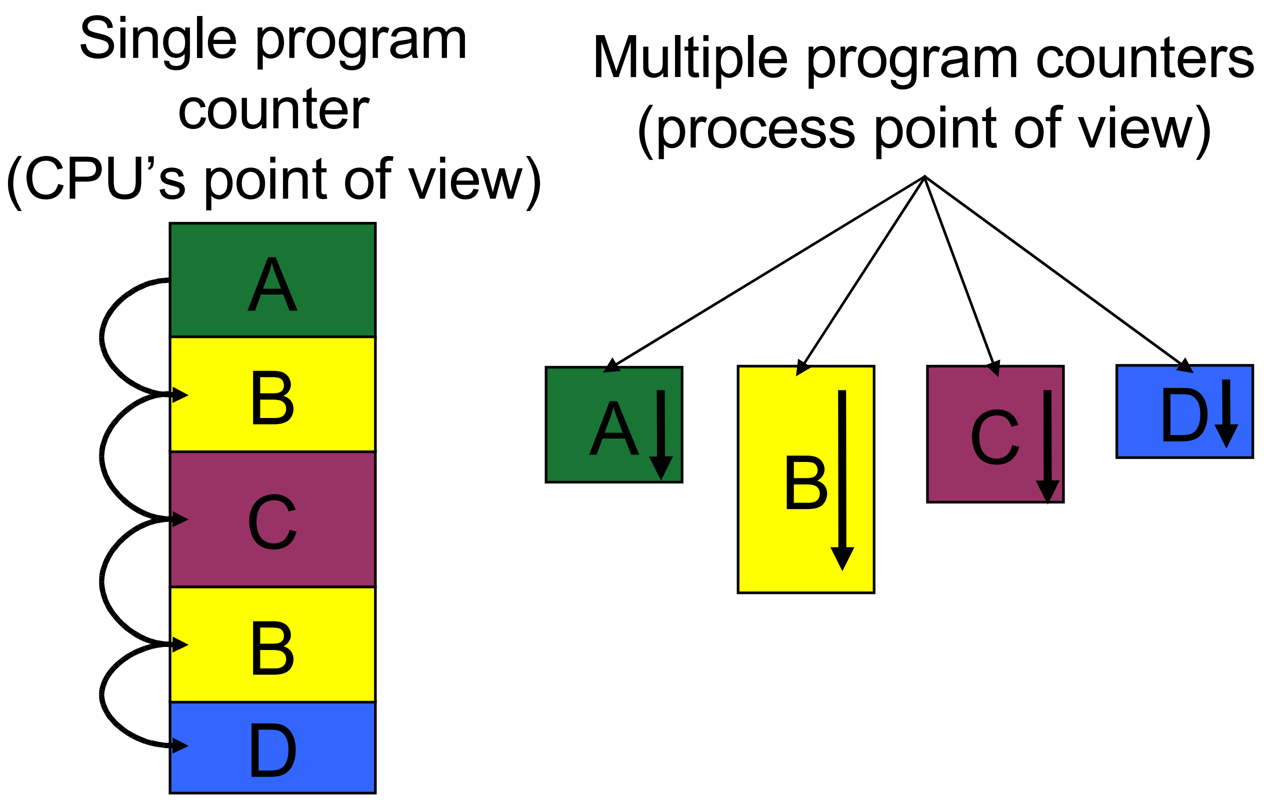
- Processes
- Threads
- Scheduling
- Interprocess communication
- Classical IPC problems

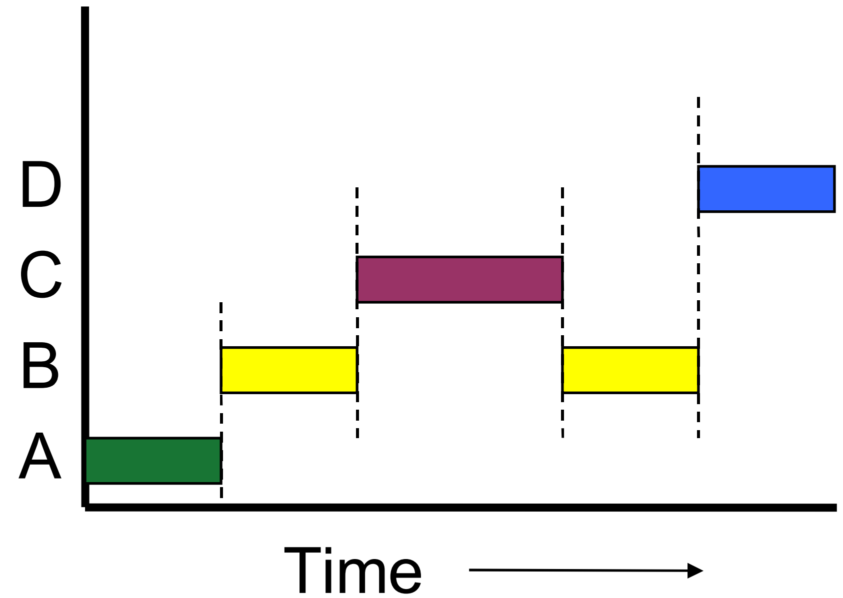
What is a process?

- 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

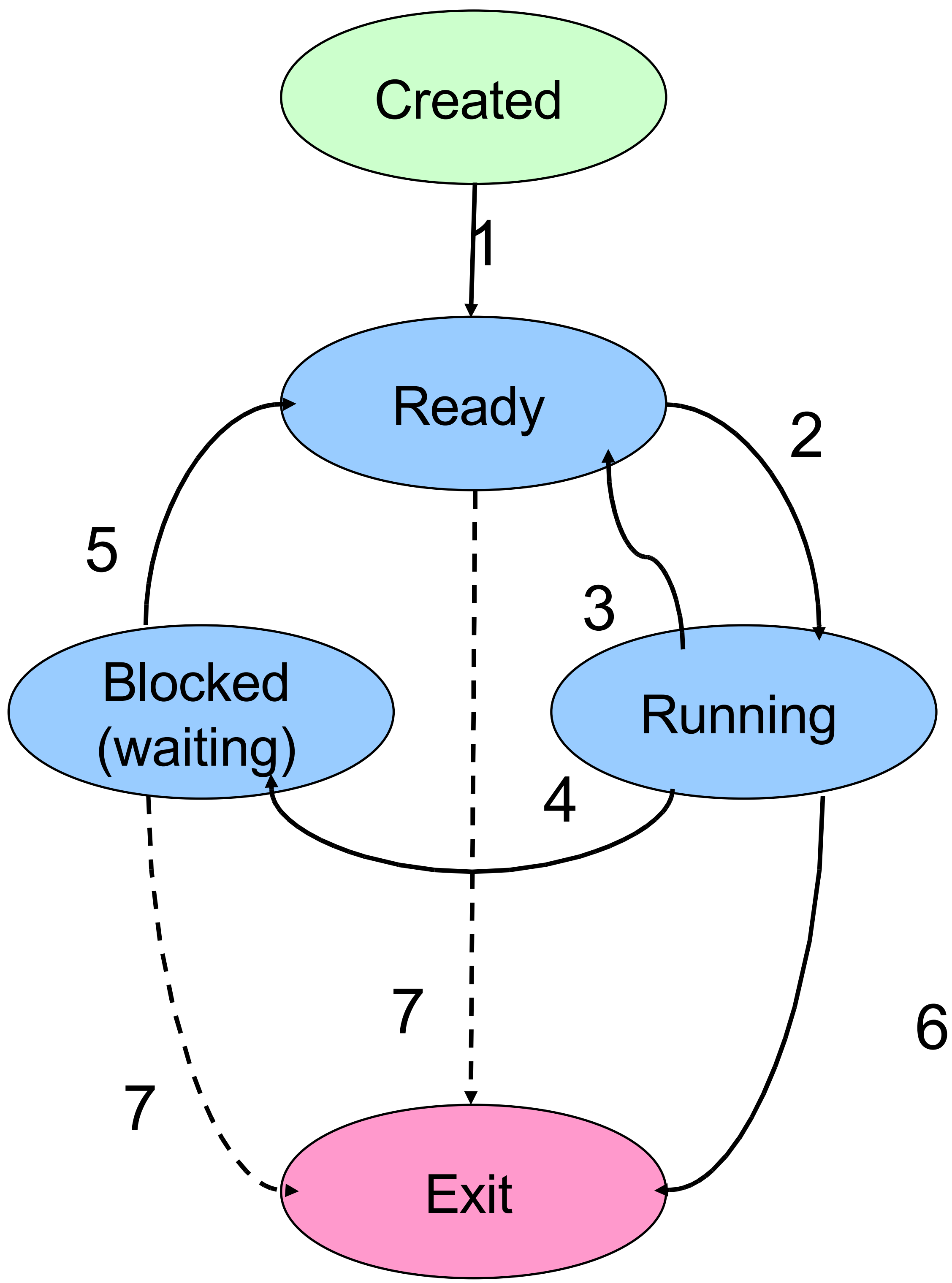
When do processes end?

- 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"

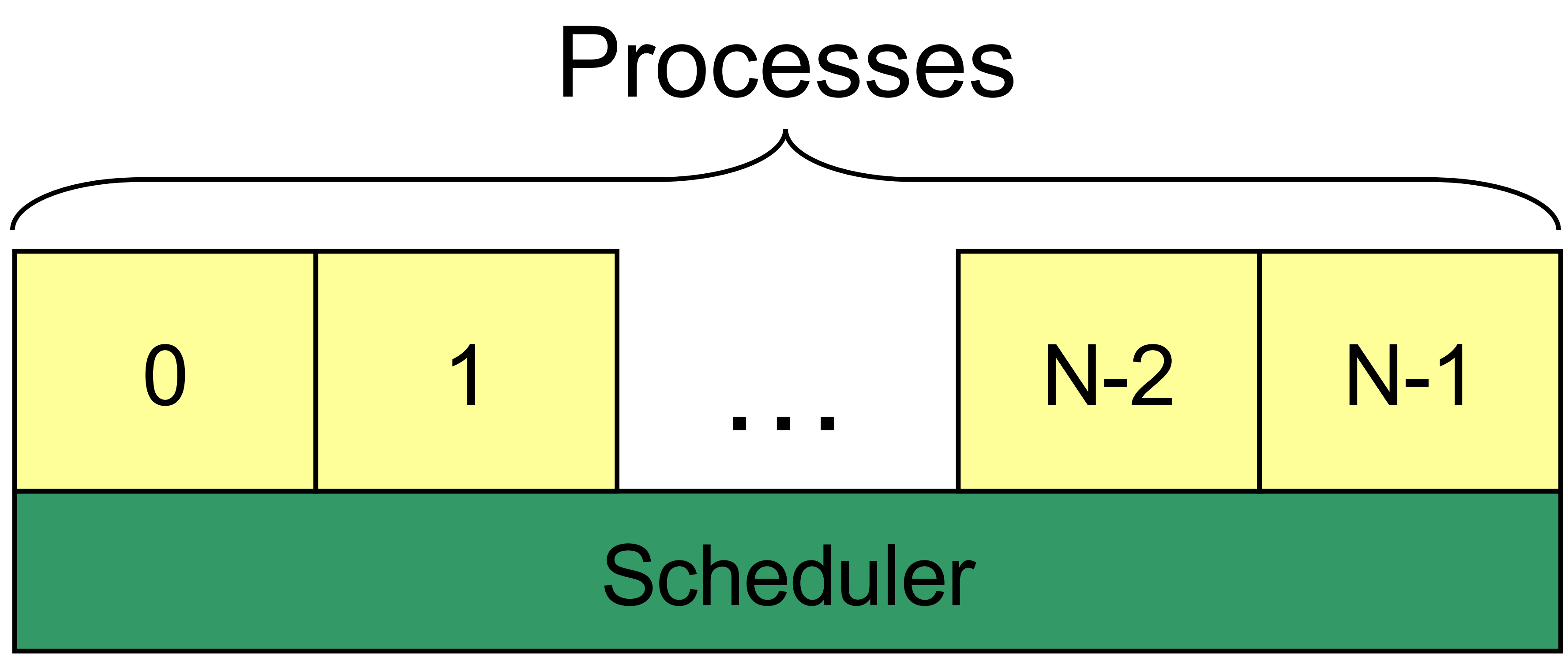
Process states



- 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?

May be stored on stack

Process management

Registers
Program counter
CPU status word
Stack pointer
Process state
Priority / scheduling parameters
Process ID
Parent process ID
Signals
Process start time
Total CPU usage

File management

Root directory
Working (current) directory
File descriptors
User ID
Group ID

Memory management

Pointers to text, data, stack
or
Pointer to page table

What happens on a trap/interrupt?

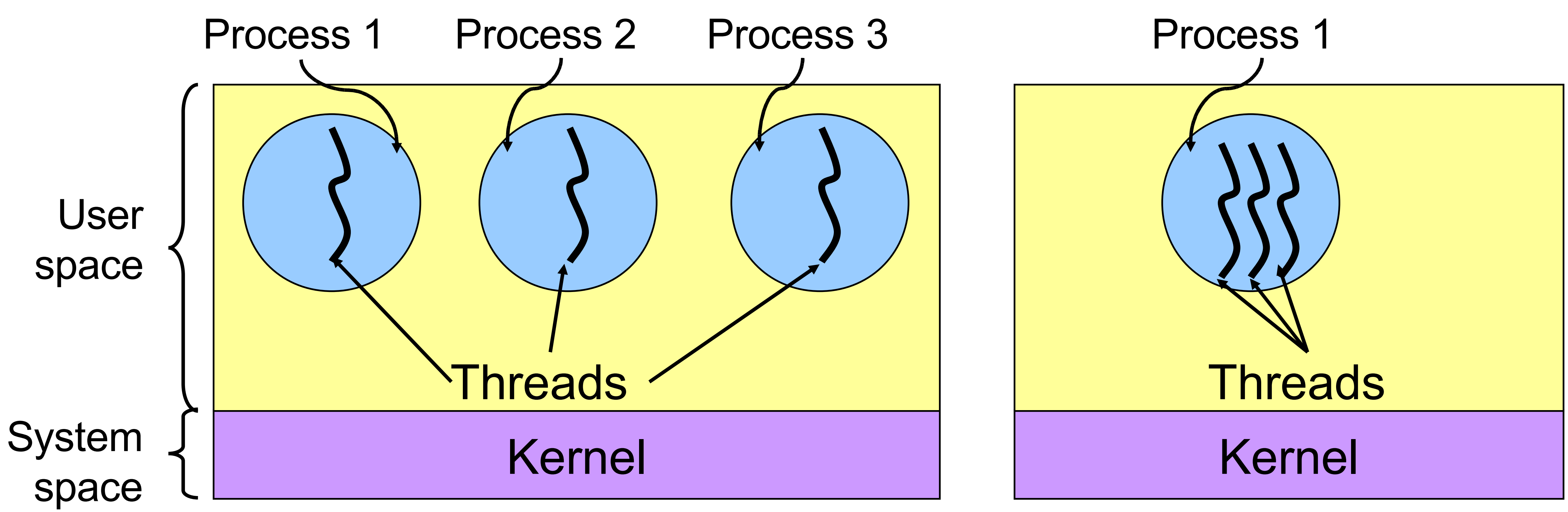
- Hardware saves program counter (on stack or in a special register)
- Hardware loads new PC, identifies interrupt

1. Assembly language routine saves registers

- Assembly language routine sets up stack
- Assembly language calls C to run service routine
- Service routine calls scheduler
- Scheduler selects a process to run next (might be the one interrupted…)
- 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



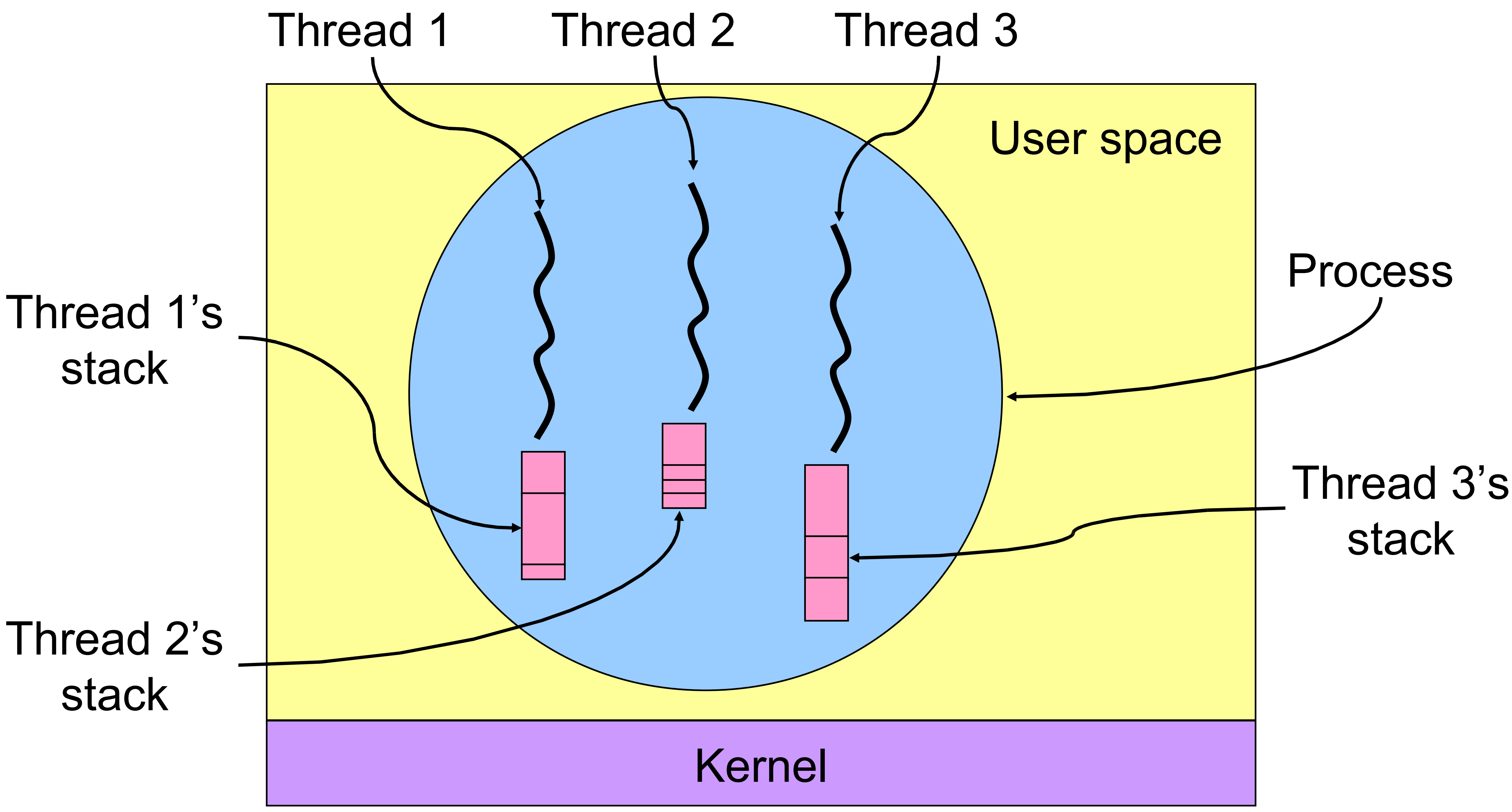
Process & thread information

Per process items

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

Per thread items	**Per thread items**	**Per thread items**
Program counter	Program counter	Program counter
Registers	Registers	Registers
Stack & stack pointer	Stack & stack pointer	Stack & stack pointer
State	State	State

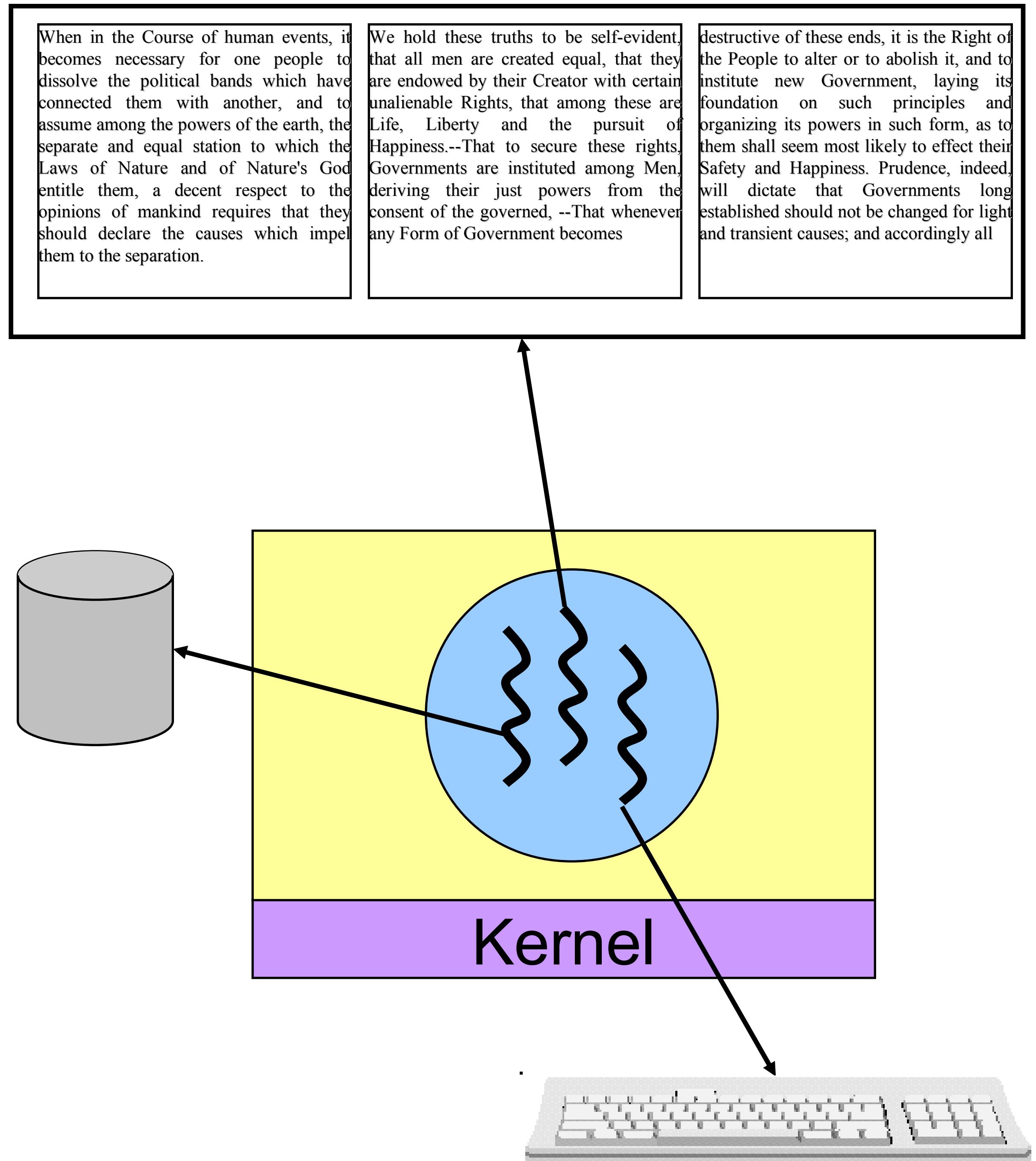
Threads & stacks



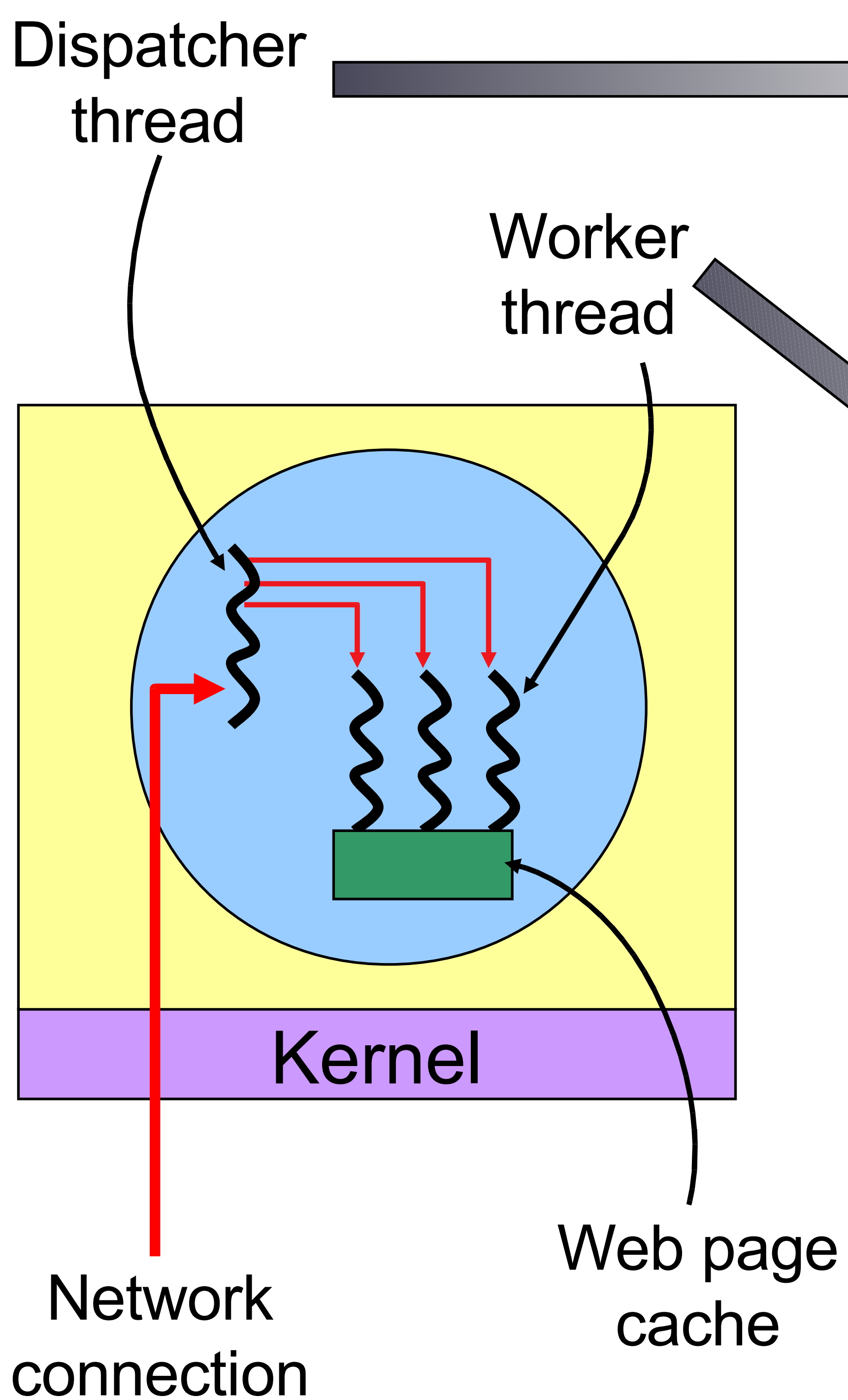
- Each thread has its own stack!

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




```
while(TRUE) {
  getNextRequest(&buf);
  handoffWork(&buf);

}
```

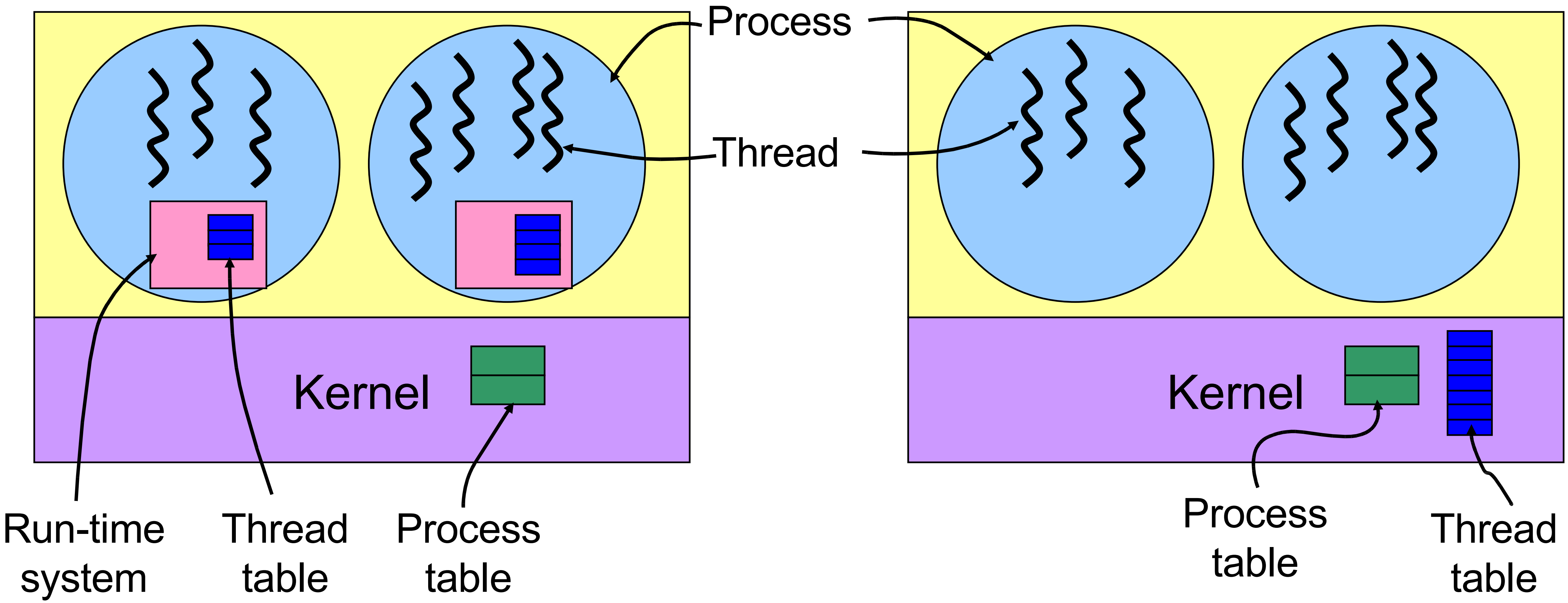
```
while(TRUE) {
  waitForWork(&buf);
  lookForPageInCache(&buf,&page);
  if(pageNotInCache(&page)) {
    readPageFromDisk(&buf,&page);
  }
  returnPage(&page);

}
```

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

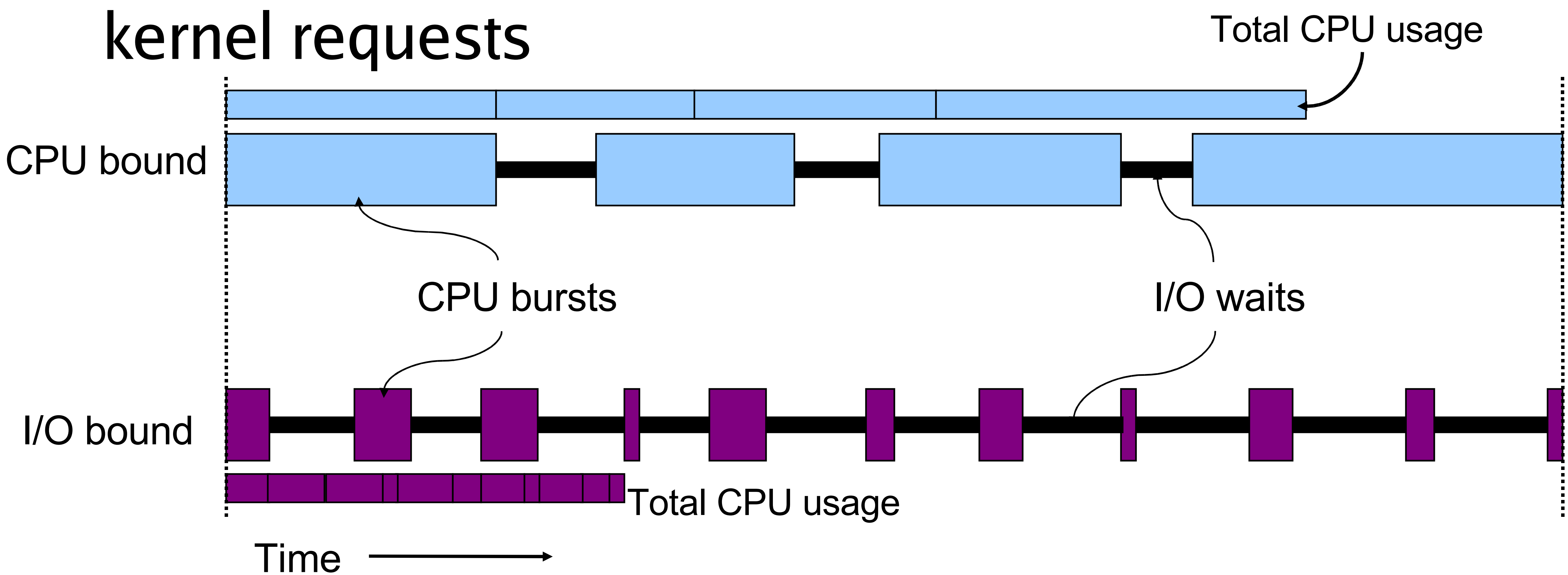
\+ More flexible scheduling
\+ Non-blocking I/O
\- Not portable

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 many I/O requests
- Other processes are I/O-bound and make many kernel requests



Off-line vs. On-linescheduling

Off-line algorithms

- Get **all** the information about **all** the jobs to schedule as their input
- Outputs the scheduling sequence
- Preemption is never needed

On-line algorithms

- Jobs arrive at unpredictable times
- Very little info is available in advance
- Preemption compensates for lack of knowledge

Using preemption

On-line short-term scheduling algorithms

- Adapting to changing conditions: e.g., new jobs arrive
- Compensating for lack of knowledge: e.g., job run-time

Periodic preemption keeps system in control

Improves fairness

  Gives I/O bound processes chance to run

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

Arrow's Theorem

- Consider 3 parameters A, B, C for which
  - 7 applns vote A > B > C
  - 6 applns vote B > C > A
  - 5 applns vote C > A > B

- So A > B desired by 7+5 applns (12)
- vs B > A by 6 applns
- Similarly, B > C voted in by 7+6 (13) vs C>B (5)
- And C > A voted in by 6+5 (11) vs A>C (7)

- So, majorities prefer A>B, B>C, C>A which together cannot be satisfied!

Interactive vs. batch scheduling

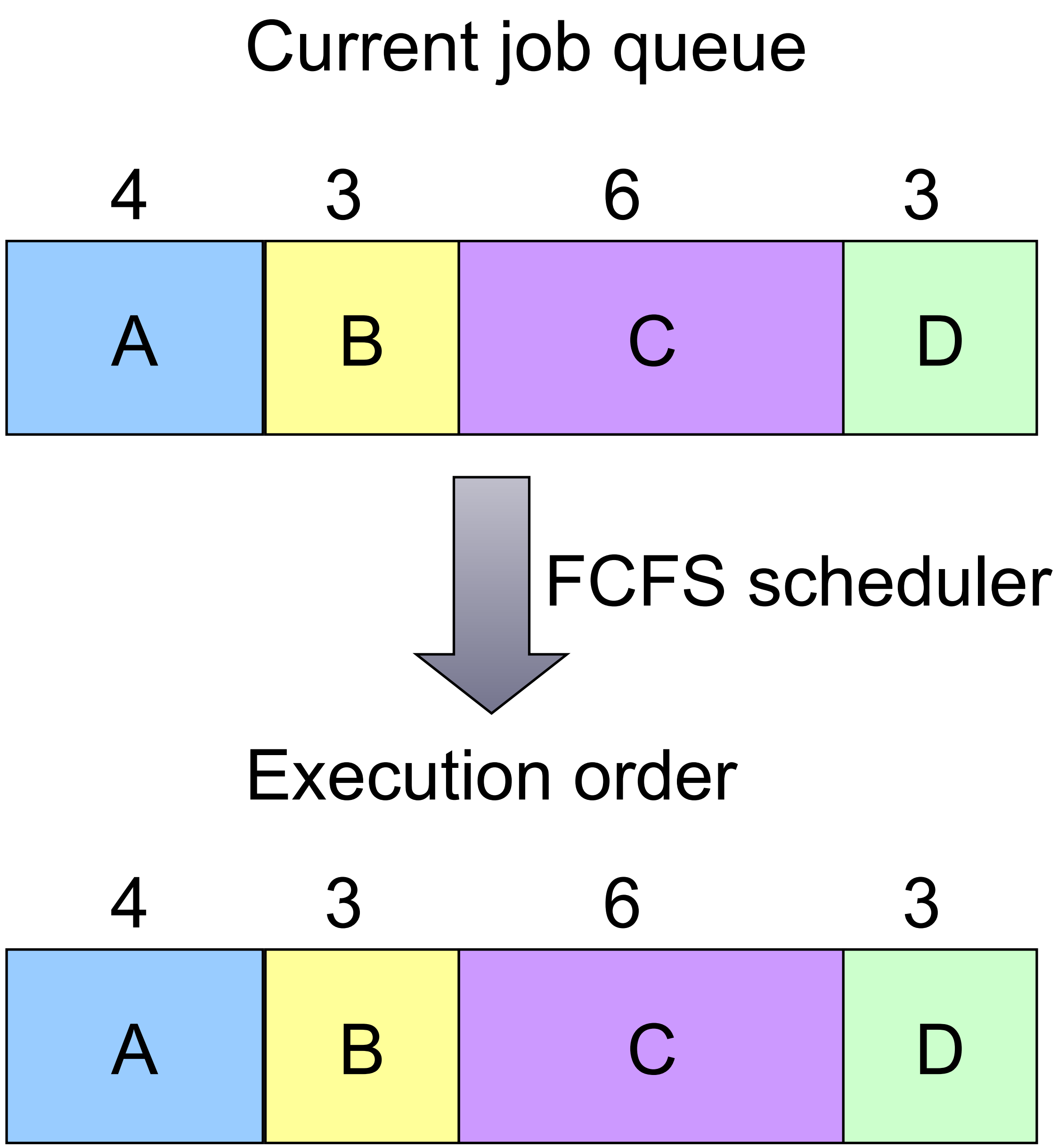
Batch

First-Come-First-Served (FCFS)
Shortest Job First (SJF)
Shortest Remaining Time First (SRTF)
Priority (non-preemptive)

Interactive

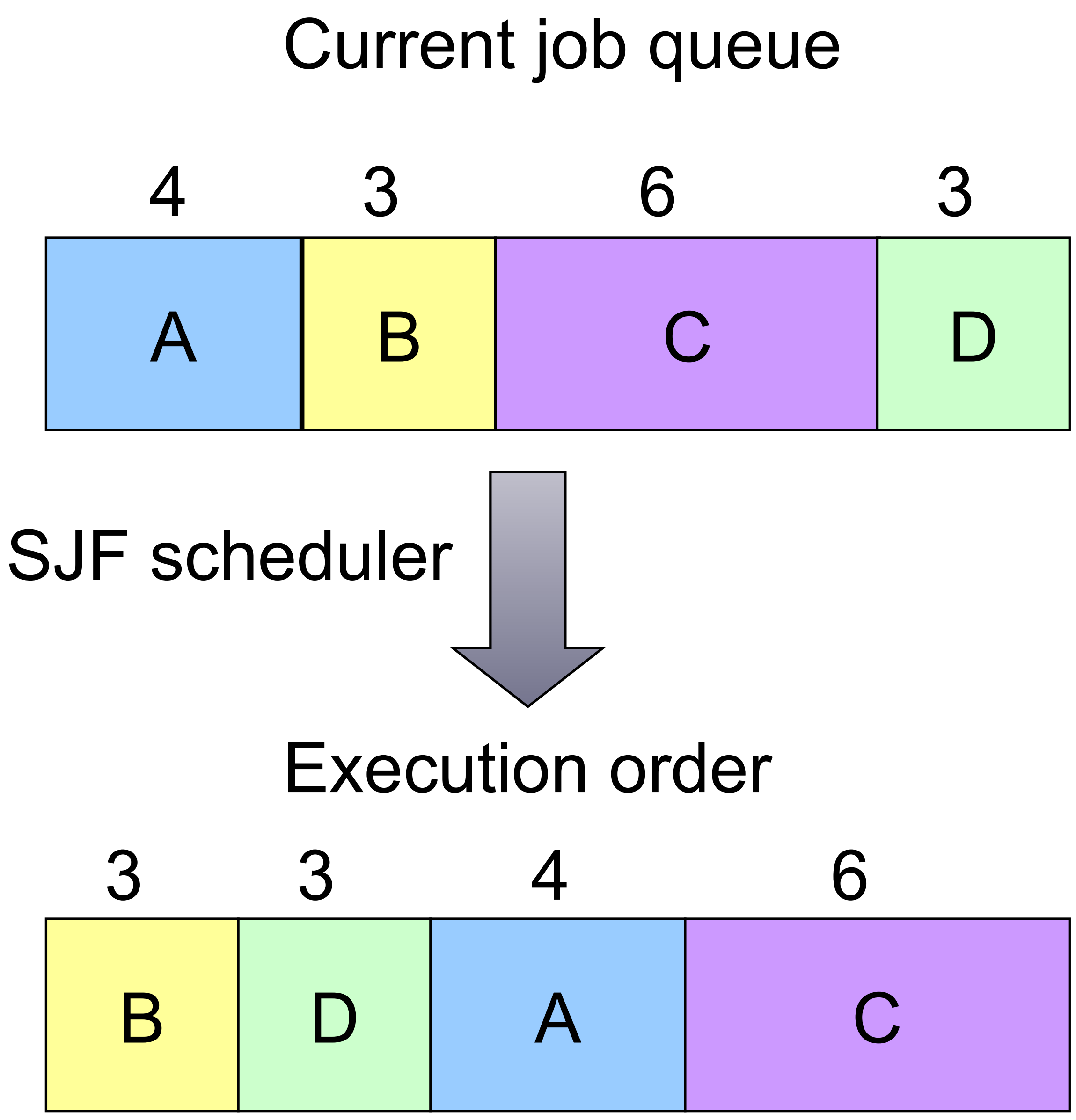
Round-Robin (RR)
Priority (preemptive)
Multi-level feedback queue
Lottery scheduling

First Come, First Served (FCFS)



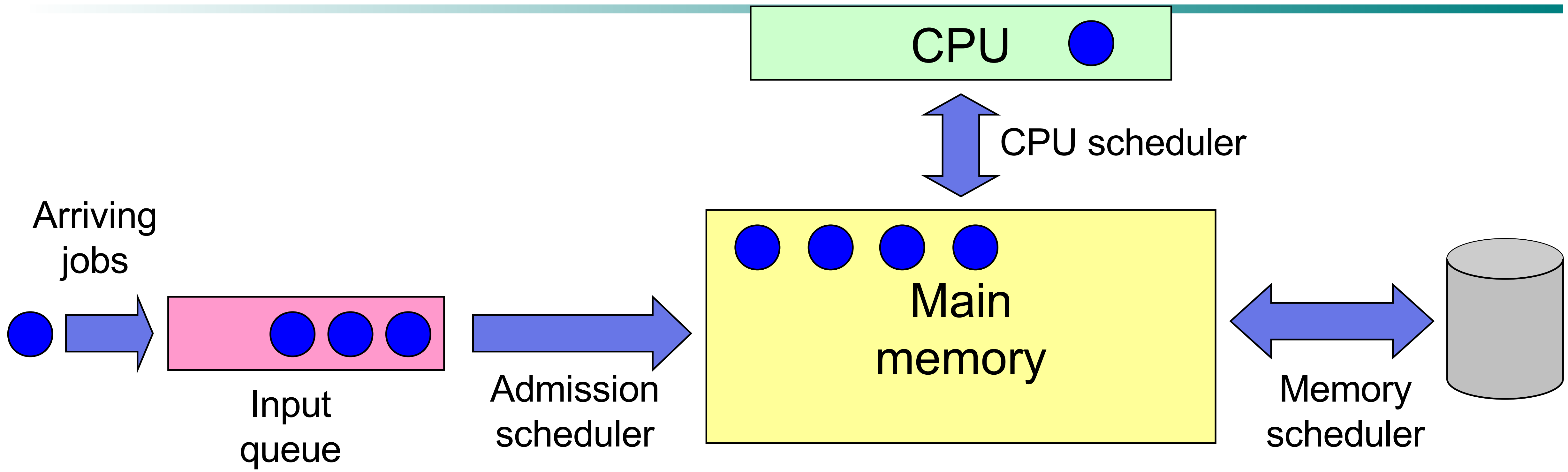
- Goal: do jobs in the order they arrive
  - Fair in the same way a bank teller line is fair
- Simple algorithm!
- Problem: long jobs delay every job after them
  - Many processes may wait for a single long job

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?

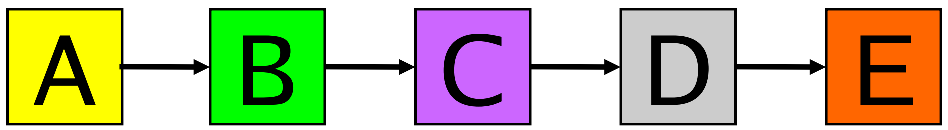
Three-level scheduling

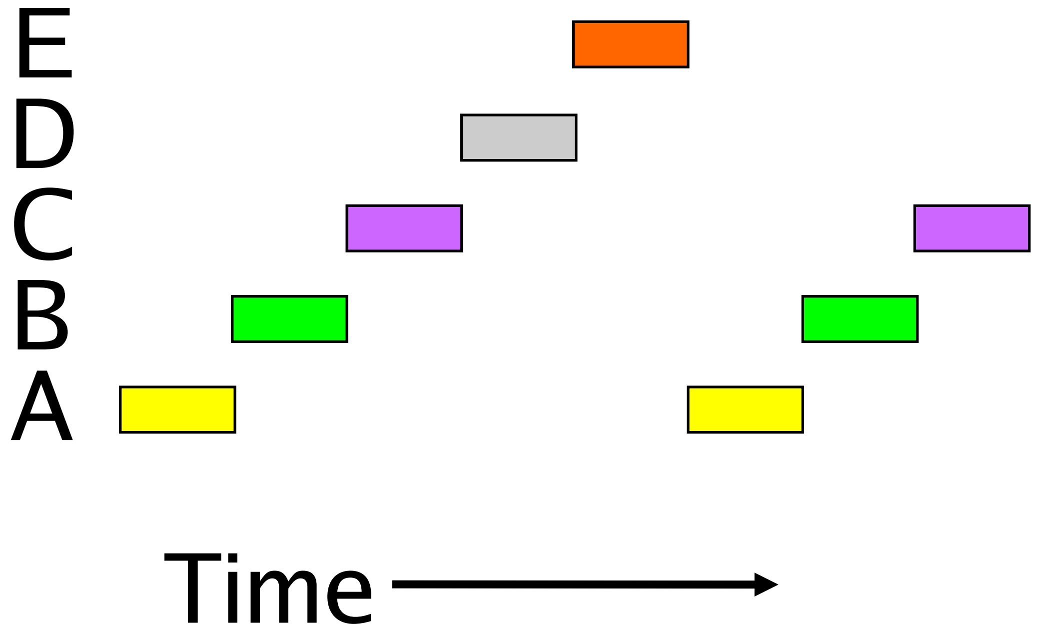


- Jobs held in input queue until moved into memory
  - Pick "complementary jobs": small & large, CPU- & I/O-intensive
  - Jobs move into memory when admitted
- CPU scheduler picks next job to run
- Memory scheduler picks some jobs from main memory and moves them to disk if insufficient memory space

Round Robin (RR) scheduling

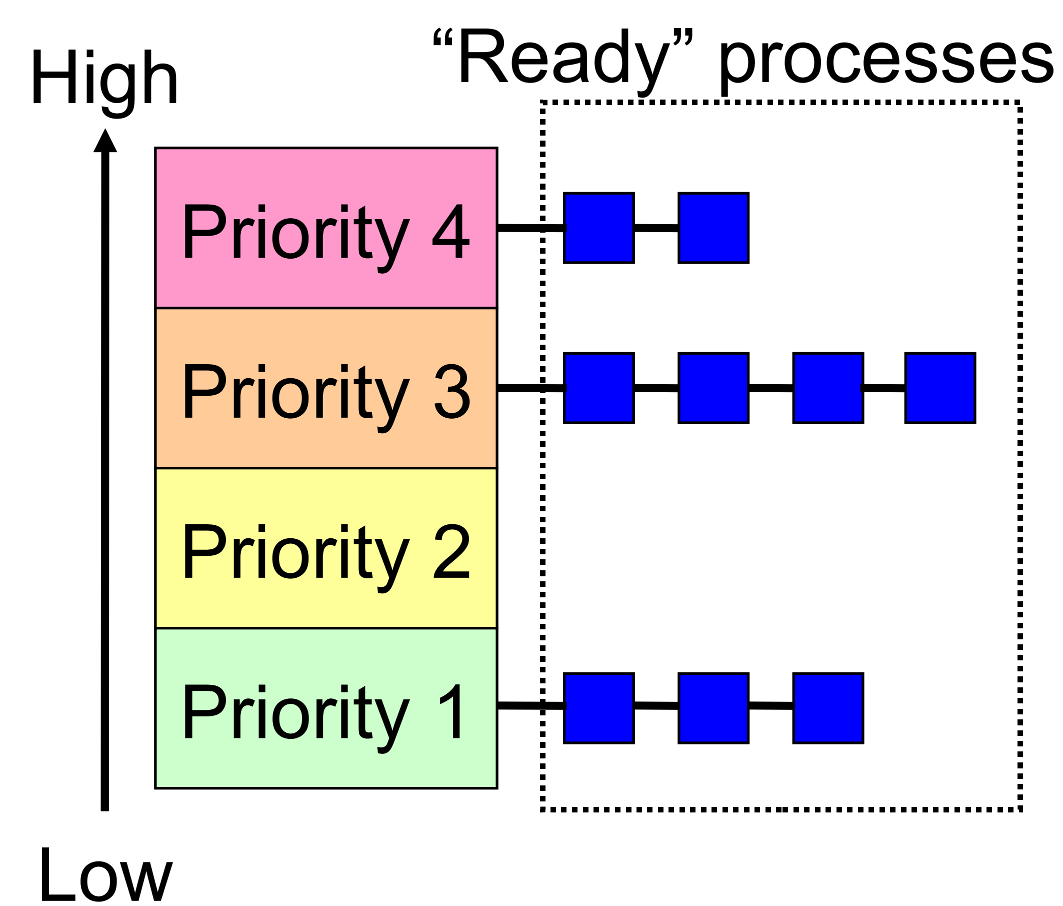
- Round Robin scheduling
  - Give each process a fixed time slot (quantum)
  - Rotate through "ready" processes
  - Each process makes some progress
- What's a good quantum?
  - Too short: many process switches hurt efficiency
  - Too long: poor response to interactive requests
  - Typical length: 10–100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



Priority Scheduling

- RR is oblivious to the process past
- I/O bound processes are treated equally with the CPU bound processes
- Solution: prioritize processes according to their past CPU usage

$$E_{n+1} = \alpha T_n + (1-\alpha)E_n, 0 \le \alpha \le 1$$

$$\alpha = \frac{1}{2} : E_{n+1} = \frac{1}{2}T_n - \frac{1}{4}T_{n-1} + \frac{1}{8}T_{n-2} + \frac{1}{16}T_{n-3} + \ldots$$

- T_n is the duration of the *n*-th CPU burst
- E_{n+1} is the estimate of the next CPU burst

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!

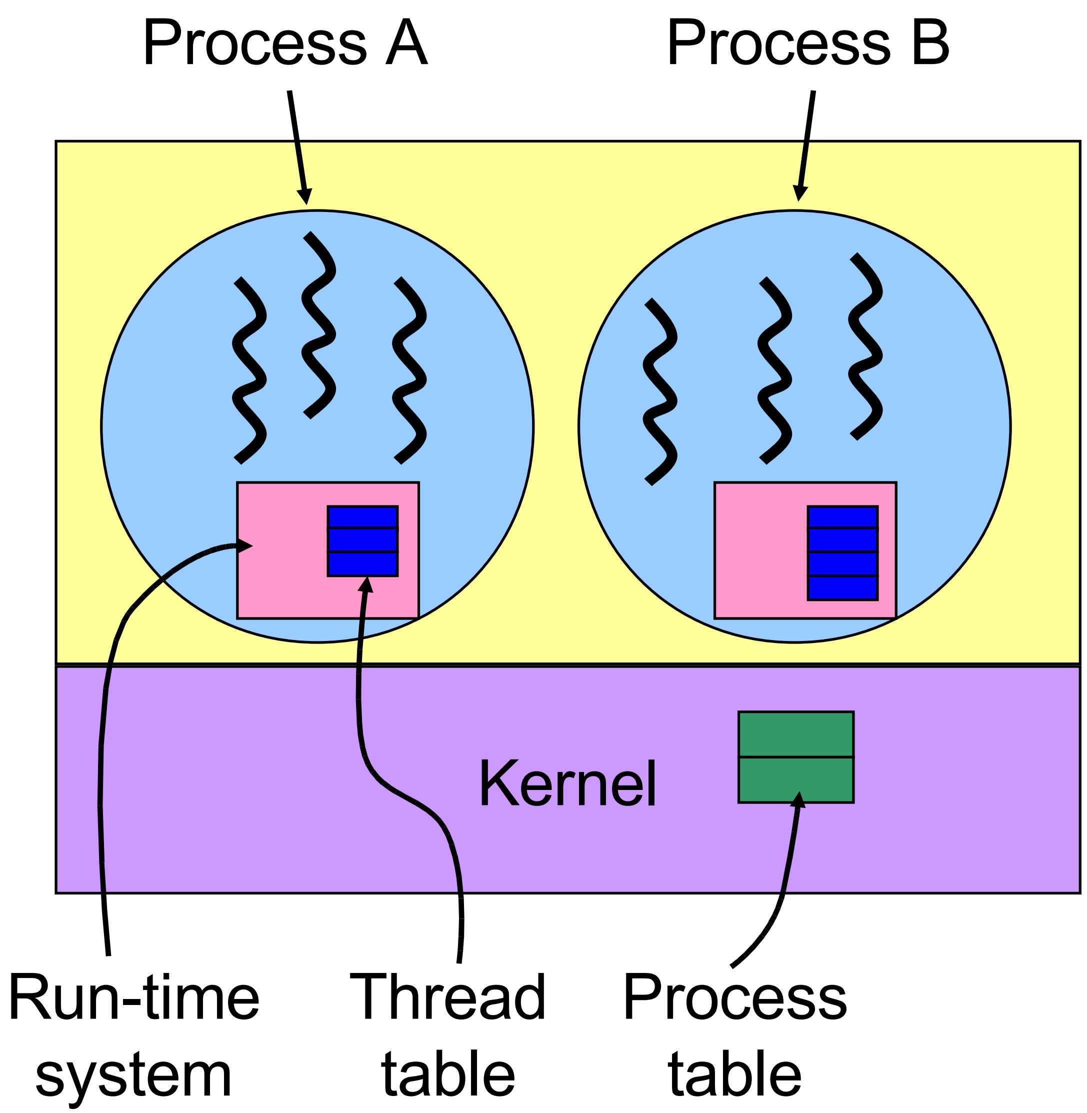
Lottery scheduling

- 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

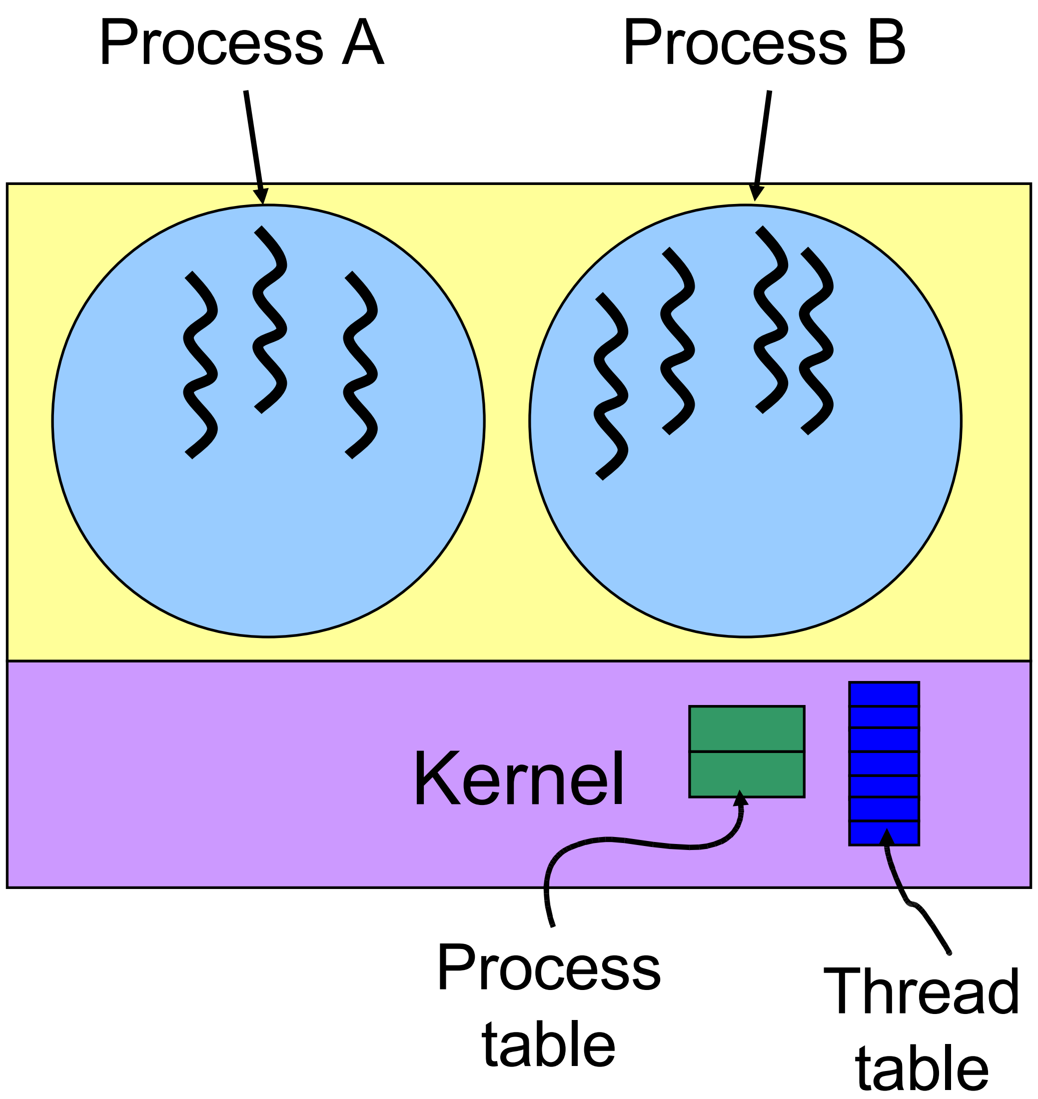
- Separate what may be done from how it is done
  - Mechanism allows
    - Priorities to be assigned to processes
    - CPU to select processes with high priorities
  - Policy set by what priorities are assigned to processes
- Scheduling algorithm parameterized
  - Mechanism in the kernel
  - Priorities 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

Scheduling still problematic!

eg: Sol 2.2. Expt: Mix of jobs with

- typing (text editor using X: interactive)
- video (RT video player: captures data from digitizer, dithers to 8b & then displays thru X: continuous media)
- compute (make appl: batch)

1st Expt: Make all jobs timesharing

- input events (mouse/kbd) not accepted, video freezes, sh does not run!
- batch class spawns and parents wait for children=> "I/O" intensive => repeated priority boosts for sleeping
- window server identified as "CPU-intensive" and priority decreases
- typing appl suffer as X does not run!

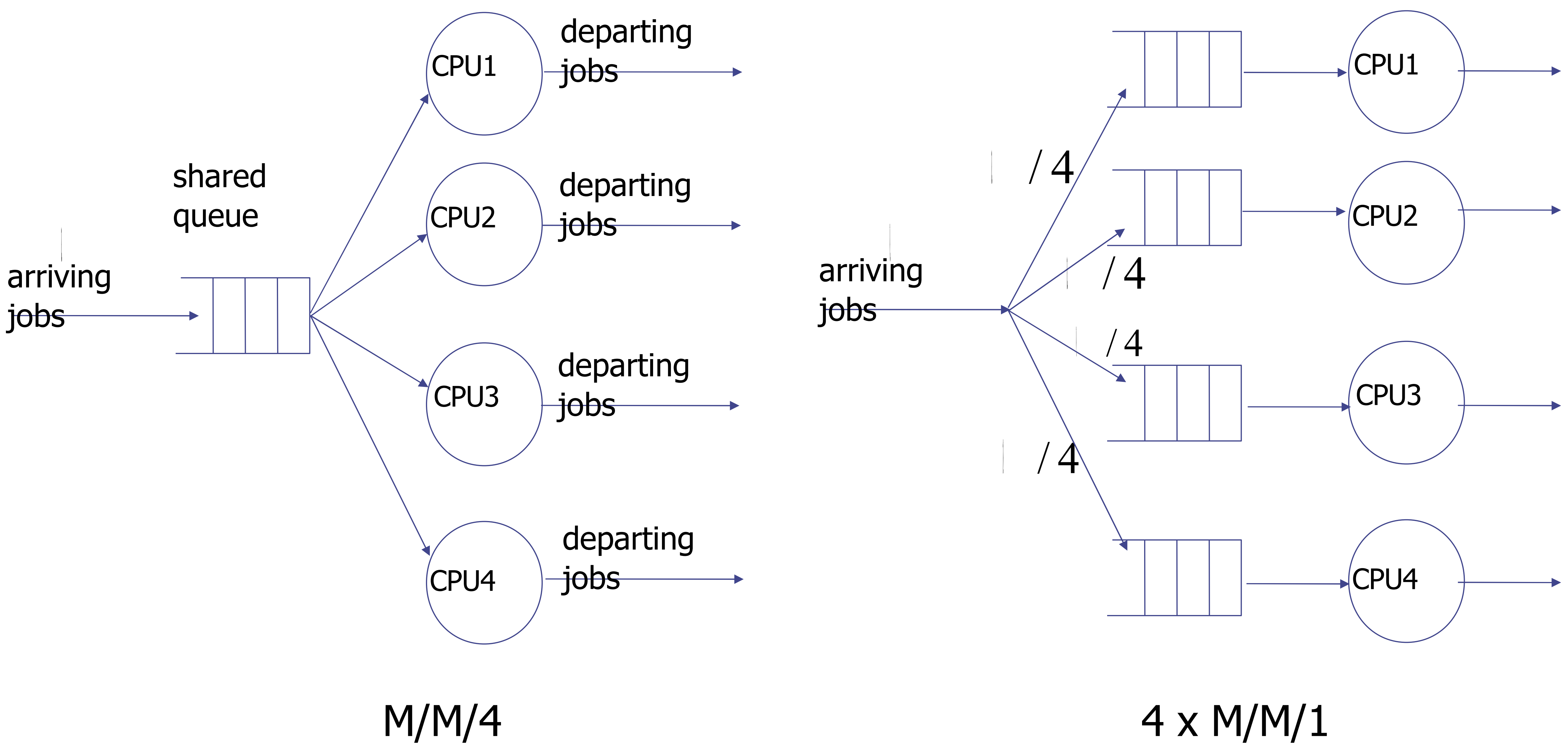
2nd Expt: assign RT to video:
input not accepted, video degrades badly
video active all the time: so TS tasks (shell, X) not run!

3rd Expt: assign X to RT:
mouse OK but batch hogs the CPU

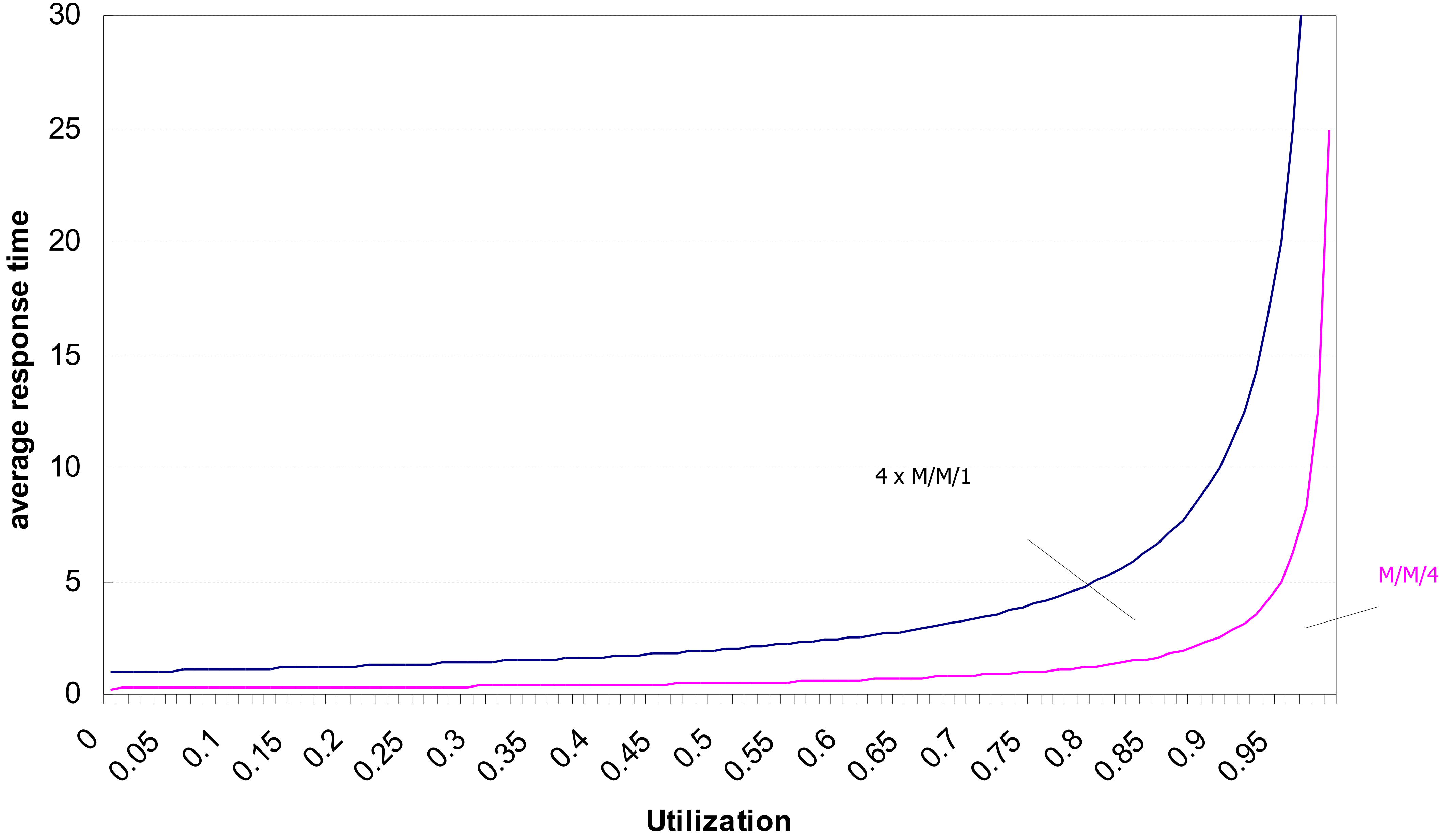
4th Expt: assign X+video to RT:
typing and batch suffer, sh does not run
flushing dirty pages to disk, process swapping do not happen!

5th Expt: X+typing+video in RT with P(X)>P(typing)>P(video):
typing does not run as it needs STREAMS!

Which is better?



M/M/4!



30
25
20
15
10
5
0
average response time
4 x M/M/1
M/M/4
0
0.05
0.1
0.15
0.2
0.25
0.3
0.35
0.4
0.45
0.5
0.55
0.6
0.65
0.7
0.75
0.8
0.85
0.9
0.95
Utilization

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;
```

Atomic statements:

$\boxed{\text{Counter += 1;}}$

$\boxed{\text{Counter -= 1;}}$

Producer

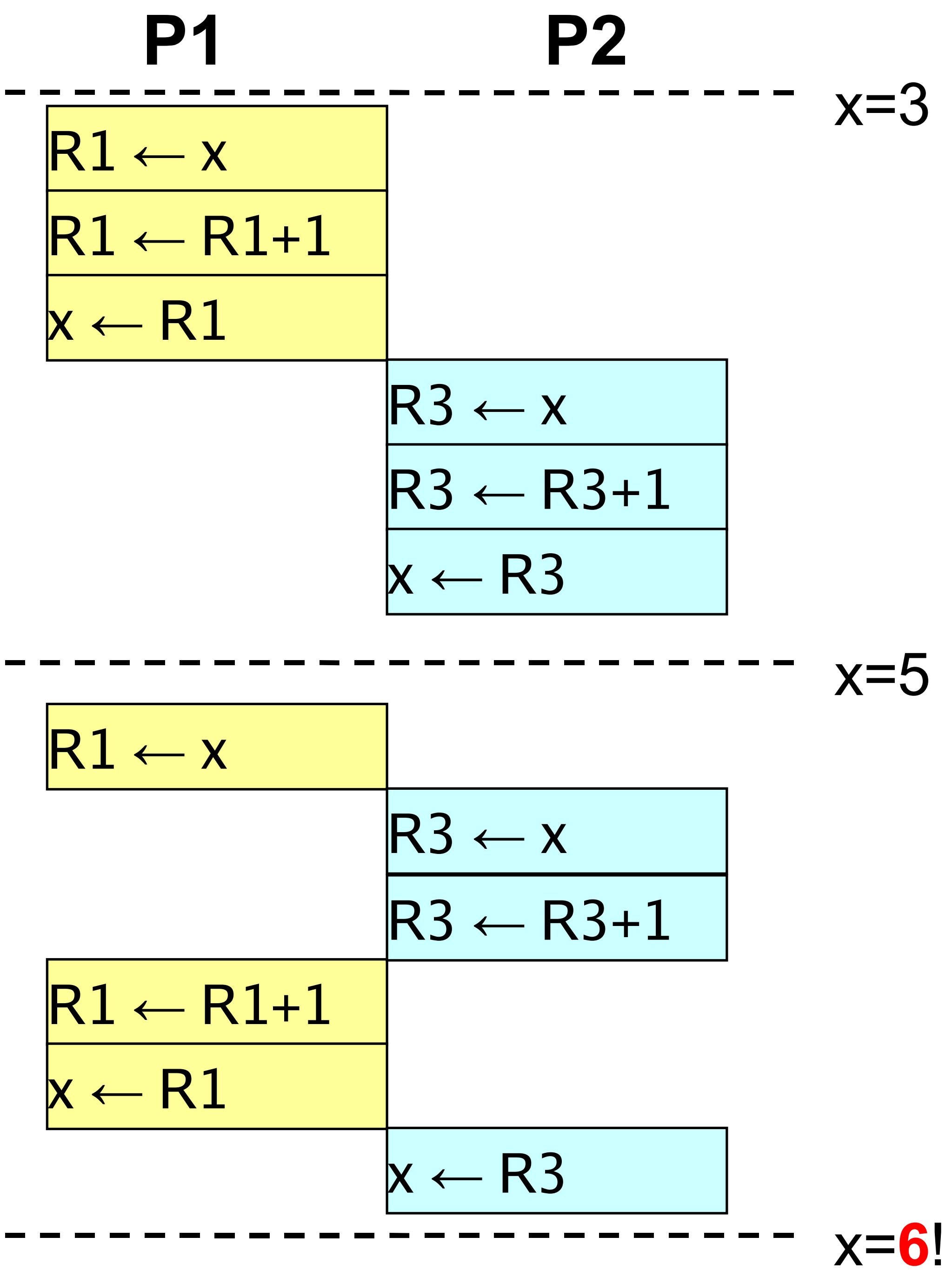
```
Item pitm;
while (1) {
  …
  produce an item into pitm
  …
  while (counter == n)
    ;
  buffer[in] = pitm;
  in = (in+1) % n;
  counter += 1;
}
```

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…



Fundamentals

- P1: y=y+1 || P2: y=y-1 ; initially y=1
- Interleaving model of concurrency

  Only result: y=1
- True concurrency model

  y can be 0,1,2:

  0: t=y; s=y-1; y=t+1; y=s

  1: P1 atomically followed by P2 or vice versa

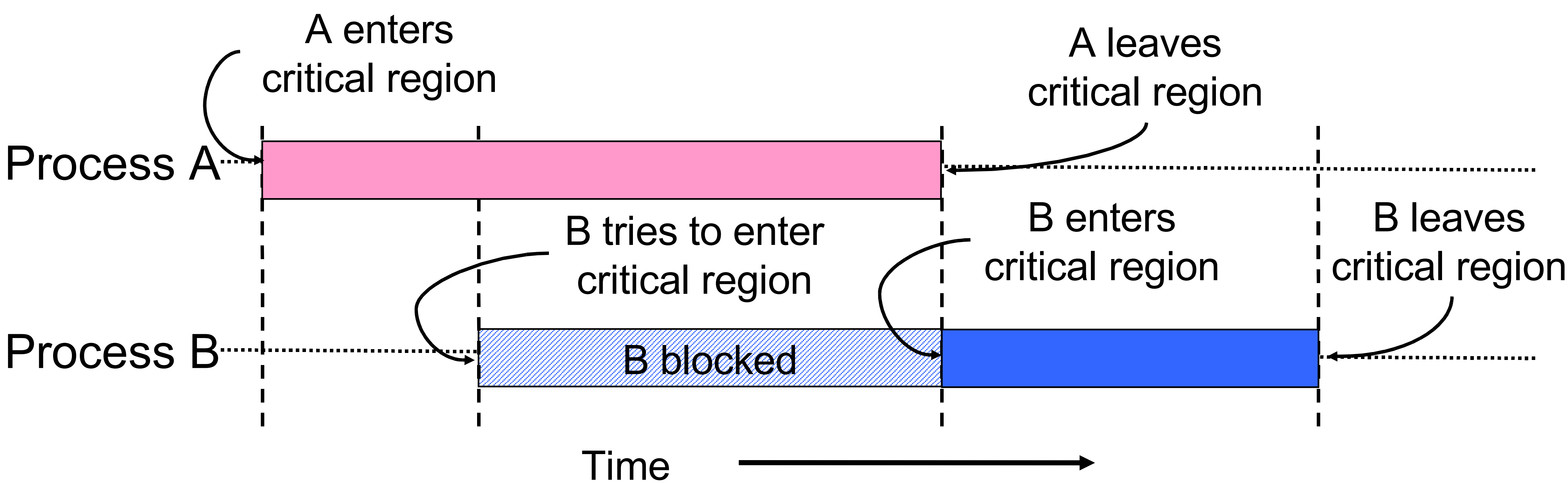
  2: t=y; s=y-1; y=s; y=t+1
- Granularity of atomic actions true reason for diff

P1: t=y; y=t+1 || P2: s=y-1; y=s

Only one "critical" reference

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



Synch

Two types of synch:

- Mutual Exclusion
- Condition synchronization

- Fine-grained synch: using HW primitives
- Coarse-grained synch: constructed atomic actions

- Properties:
  - Mutual Exclusion (safety)
  - Absence of deadlock or progress (liveness)
  - Absence of unnecessary delay (safety)
    - Should depend only on processes trying to enter
  - Eventual Entry (liveness)
  - Sometimes also worry about “bounded wait”

Terms

- Atomic actions; Critical sections; history; trace
- Deadlock: no transitions to other states
- Livelock: "busy loop": transitions to other states but does not change anything
- Safety: nothing bad happens
  - Eg: partial correctness
- Liveness: something good happens eventually
  - Eg: termination;
- Every property can be formulated in terms of safety and liveness properties
  - total correctness: safety+liveness

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
}
```

Mutual Exclusion

```
P: while{
        Entry protocol
        Critical section
        Exit protocol
        Non-critical section
    }    =>
```

```
bool in1=in2=false;
// Mutex: ~(in1 && in2)
P1: while(1) {
        in1=true;
        <Critical section;>
        in1=false;
        <Non-critical section;>
    }
P2: ...
```

```
bool in1=in2=false;
// Mutex: ~(in1 && in2)
P1: while(1) { // Mutex && ~in1
        <await ~in2 --> in1=true;>
        // Mutex && in1
        <Critical section;>
        in1=false;
        <Non-critical section;>
        // Mutex && ~in1
    }
P2: ...
```

```
<await ~in2 --> in1=true;>:
    while (in2); in1=true  ? no Mutex
    in1=true; while (in2)  ? deadlock
<await !lock --> lock=true>
    TS(lock, cc):
     <cc=lock; lock=true>
```

Peterson's alg

Break deadlock by a tie: let last be the one that entered cs last if both interested

```
bool in1=in2=false;
// Mutex: ~(in1 && in2)
P1: while(1) { // Mutex && ~in1
      in1=true; last=1;
      <await ~in2 or last=2>
      // Mutex && in1
      <Critical section;>
      in1=false;
      <Non-critical section;>
      // Mutex && ~in1
   }
```

```
bool in1=in2=false;
// Mutex: ~(in1 && in2)
P1: while(1) { // Mutex && ~in1
      in1=true; last=1;
      while(in2 && last=1);
       // Mutex && in1
      <Critical section;>
      in1=false;
      <Non-critical section;>
      // Mutex && ~in1
   }
```

Both P1 and P2
- read/write last (multi-writer)
- write their own in but read other's

Subtlety in Peterson's alg

Note that if last=1; followed by
in1=true;
INCORRECT!
COMPILER OPTS!

```
P1: 0 while(1) {
      1   <Non- Critical section;>
      2   last=1;
      3   in1=true;
      4   while(in2 && last=1);
      5   <Critical section;>
      6   in1=false;
    }
```

Let state of 2 processes be
(PC1, PC2, last, in1, in2)

(PC1,	PC2,	last,	in1,	in2)		[last,in1/2]
(0,	0,	1,	0,	0)		
(1,	0,	1,	0,	0)		
(1,	1,	1,	0,	0)		
(2,	1,	1,	0,	0)		
(2,	2,	1,	0,	0)		
(3,	2,	1,	0,	0)		[1,1,0]
(3,	3,	2,	0,	0)		[1,1,1]
(3,	4,	2,	0,	1)		[2,1,1]
(3,	5,	2,	0,	1)		PC2 blocked
(4,	5,	2,	1,	1)	!!!	
(5,	5,	2,	1,	1)		
INCORRECT code						CORRECT

Why? in1=in2=0

P1	P2
last=1	
	last=2
	In2=1
In1	=> CS
=1 => CS	

n-process solution

- Use 2-process solution as basis
- Entry protocol loops thru n-1 stages
- Each stage uses a 2-process solution to determine winner
  - in[i] indicates which stage P[i] is executing
  - last[j] indicates which process was last to begin stage j
- Atmost n-i processes past i^{th} stage
- Solution is livelock-free, avoids unnecessary delay, ensures eventual entry
- $O(n^2)$
- Bounded numbers
  - No large #
- Note perf bug; should be for j = 1 to n-1

```
int in[1:n] = ([n] 0), last[1:n] = ([n] 0);

process CS[i = 1 to n] {
  while (true) {
    for [j = 1 to n] {          /* entry protocol */
      /* remember process i is in stage j and is last */
      last[j] = i; in[i] = j;
      for [k = 1 to n st i != k] {
        /* wait if process k is in higher numbered stage
           and process i was the last to enter stage j */
        while (in[k] >= in[i] and last[j] == i) skip;
      }
    }
    critical section;
    in[i] = 0;                  /* exit protocol */
    noncritical section;
  }
}
```

Figure 3.7 The n-process tie-breaker algorithm.

Ticket Alg

Get a ticket # larger than any prev wait till ticket # is next

TICKET Invariant: P[i] in cs =>

- turn[i]=next &
- All non-0 values of turn unique: for all i, j: 1<=i, j<=n, j != i: turn[i]=0 or turn[i] != turn[j]

Bakery algorithm: instead of above ticket alg, sets its number 1 larger than any existing & waits till it is smallest

BAKERY Invariant: P[i] in cs => turn[i] !=0 & for all j: 1<=j<=n, j!=i: turn[j]=0 or turn[i]<turn[j]

Problem: unbounded numbers (next, number)

Need fetch&add for a fine grained solution

- FA(var, incr): <temp=var; var+:=incr; return temp>

Also thru test&set

- Critical section entry
- turn[i]=number
- Number+=1
- Critical section exit

```
int number = 1, next = 1, turn[1:n] = ([n] 0);
## predicate TICKET is a global invariant (see text)

process CS[i = 1 to n] {
  while (true) {
    ⟨turn[i] = number; number = number + 1;⟩
    ⟨await (turn[i] == next);⟩
    critical section;
    ⟨next = next + 1;⟩
    noncritical section;
  }
}
```

Figure 3.8 The ticket algorithm: Coarse-grained solution.

```
int number = 1, next = 1, turn[1:n] = ([n] 0);

process CS[i = 1 to n] {
  while (true) {
    turn[i] = FA(number,1);          /* entry protocol */
    while (turn[i] != next) skip;
    critical section;
    next = next + 1;                 /* exit protocol  */
    noncritical section;
  }
}
```

Figure 3.9 The ticket algorithm: Fine-grained solution.

```
int turn[1:n] = ([n] 0);
## predicate BAKERY is a global invariant -- see text

process CS[i = 1 to n] {
  while (true) {
    ⟨turn[i] = max(turn[1:n]) + 1;⟩
    for [j = 1 to n st j != i]
      ⟨await (turn[j] == 0 or turn[i] < turn[j]);⟩
    critical section;
    turn[i] = 0;
    noncritical section;
  }
}
```

Figure 3.10 The bakery algorithm: Coarse-grained solution.

Fine Grained 2-process Bakery

turn1=turn2=0
P1:
while (1) {
  turn1=turn2+1
  while (turn2!=0 & turn1>turn2);
   critical section
   turn1=0
   non-critical section
}
P2:...

Prob: both P1 & P2 in c.s with init.

Soln: tie break: set condition to turn2>=turn1 in P2

Prob: race condition

P1 reads turn2=0 => cs

P2 reads turn1=0, sets turn2=1 => cs

Soln: each process sets $turn_i$ to *any value >=1*

while (1) {
  turn1=1; turn1=turn2+1
  while (turn2!=0&turn1>turn2);
   critical section
   turn1=0
   non-critical section
}

Why?

- turn1=1 turn2=1
- turn1=turn2+1 turn2=turn1+1
- while(turn2!=0 & turn1>turn2) while(turn1!=0 & turn2>=turn1)
- CS CS

- Without turn1=1 and turn2=1 in the beginning (both 0 in the beginning):
- turn2+1 turn2=turn1+1
- => CS
- turn1=
- => CS

Symmetric Bakery

- P1 in cs: (turn1>0) & (turn2=0 or turn1<=turn2)
- P2 in cs: (turn2>0) & (turn1=0 or turn2<   turn1)
- Use lexicographic order to make it symmetric
  - turn1>turn2 in P1   --> (turn1,1) > (turn2,1)
  - turn2>=turn1 in P2 --> (turn2,2) > (turn1,1)
- n-process solution: Global turn[1:n] = 0
- BAKERY: P[i] in cs --> forall j: 1<=j<=n, j<>i: turn[j]=0 or turn[i]<turn[j] or (turn[i]=turn[j] & i<j)

```
P[i]: while (1) {
      turn[i]=1; turn[i]= max(turn[1..n])+1;
      forall j:1..n st i<>j  while ((turn[j]<>0) & (turn[i],i)>(turn[j], j));
      cs; turn[i]=0; non-cs
}  // Note that there is no guarantee that 2 processes do not get same #
```

```
int turn[1:n] = ([n] 0);

process CS[i = 1 to n] {
  while (true) {
    turn[i] = 1; turn[i] = max(turn[1:n]) + 1;
    for [j = 1 to n st j != i]
      while (turn[j] != 0 and
                (turn[i],i) > (turn[j],j)) skip;
    critical section;
    turn[i] = 0;
    noncritical section;
  }
}
```

Figure 3.11 Bakery algorithm: Fine-grained solution.

Properties of Bakery

Requires that a process be able to read a word of memory while another process is writing it

- variables read by multiple processes, but written by only a single process: good for dcs!

Bakery alg works regardless of what value is obtained by a read that overlaps a write

- only the write must be performed correctly. The read may return any arbitrary value (a *safe* register)
- implements mutual exclusion without relying on any lower-level mutual exclusion
- <u>reading and writing need not be atomic ops</u>
- Before this algorithm, it was believed that mutual exclusion problem unsolvable without using lower-level mutual exclusion (infinite regress!!!)

Bakery algorithm for many processes

- Notation used
  - <<< is lexicographical order on (ticket#, process ID)
  - (a,b) <<< (c,d) if (a <<< c) or ((a==c) and (b<d))
  - Max(a_0,a_1,…,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
}
```

Lamport's Bakery alg for n processes

R

Process P_i

do {

T

D

$choosing[i] = true;$

$number[i] = \max(number[0], number[1], ..., number[n-1]) + 1;$

$choosing[i] = false;$

T-D

$for (j = 0; j < n; j++) \{$

$\quad while (choosing[j]);$

$\quad while ((number[j] != 0) \&\& ((number[j], j) < (number[i], i)));$ t1 t2

}

C

critical section S

E

$number[i] = 0;$

remainder section

} while(1)

Structure of the algorithm

R - code prior to using Bakery algorithm

T - creating of a ticket and awaiting for permission to enter critical section

D - creation of a number (first part of a ticket)

T-D - awaiting for the permission to enter critical section

C - critical section itself

E - code executed upon exit from critical section

Why is t1 *while(choosing[j]);* there?

Consider 2 procs 1&2, both in D. max(...) is computed, assigned to proc 2 but not to proc 1 yet. (Both will get the same value at end of D.) Proc 2 enters CS *if t1 is not there.* Now proc 1 gets assigned and it will also enter CS as not (number[2], 2) < (number[1], 1)

Remark on Fairness

- A process that obtained a ticket (number[k],k) will wait at most for (n-1) turns, when other processes will enter the critical section
- For example if all the processes obtained their tickets at the same time they will look like (q,1),(q,2)…(q,n)

  In which case process P_n will wait for processes $P_1 \ldots P_{n-1}$

  to complete the critical section

Bakery alg satisfies "FIFO after a wait-free doorway" fairness property: if Pi completes doorway before Pj enters T, then Pj cannot enter C before Pi. However, it is not FIFO based on time of entry, etc.

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
}
```

Test & Set

TS(lock, cc)
<cc=lock; lock=true>

- Lock:
  - Repeat
  -   TS(lock, cc)
  - Until !cc
- Unlock:
  - lock=false

- Too much contention for the bus!

Test & Test & Set

- Lock:
  - Repeat
  -   while (lock);
  -   TS(lock, cc)
  - Until !cc

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
    - Down(S): while (S<=0) {}; S-= 1;
    - Up(S): S+=1;
  - Down() is another name for P()
  - Up() is another name for V()
  - Modify implementation to eliminate busy wait from 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_0	**Process P_1**
.	.
.	.
.	.
`// Execute code for A`	`flag.down ();`
`flag.up ();`	`// 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

Have we solved the problem?

- P() and V() must be executed atomically
- In uniprocessor system may disable interrupts
- In multi-processor system, use hardware synchronization primitives such as TS, FAA
- Involves some limited amount of busy waiting

RMW shared variables

Test&set

$f(test\&set, v) = (v, 1)$

Swap

$f(swap(u), v) = (v, u)$

Fetch& add

$f(fetch\&add(u), v) = (v, v+u)$

Compare & swap

$f(CAS(u,v), w)$

(w,v) if $u=w$

(w,w) otherwise

LoadLinked & Store Conditional

Load-linked/Store-Conditional

Exchange what is in R4 with (R1)

```
Try:MOV R4, R3   //R3=R4
    LL (R1), R2
    SC R3, (R1)  // if no mem op in betw, SC returns 1 else 0 in R3
    BEQZ R3, Try
    MOV R2, R4
```

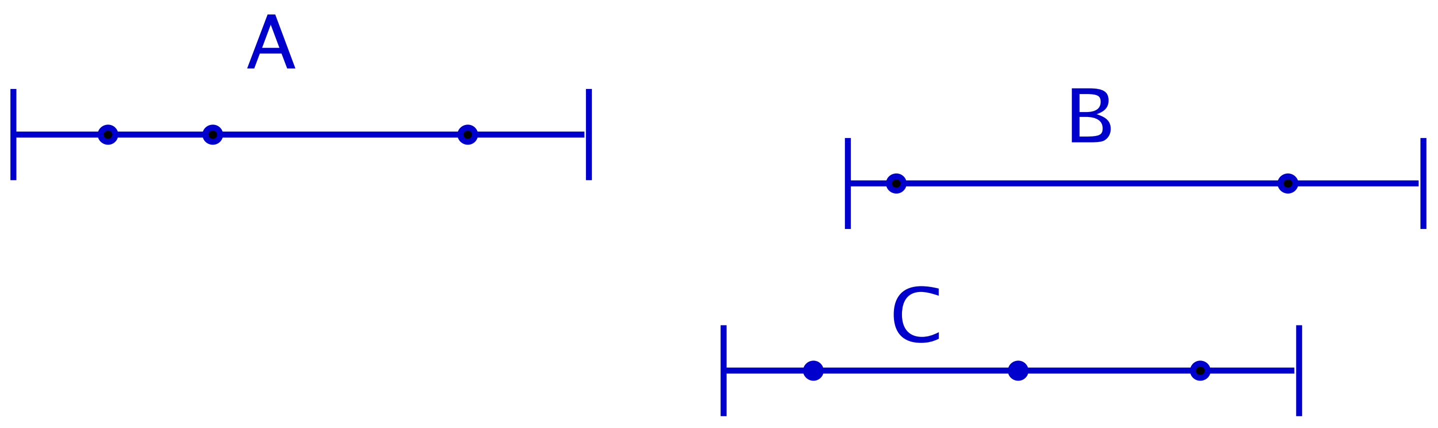
Incr a memory value atomically:

```
Try:LL (R1), R2
    INCR R2, R3
    SC R3, (R1)
    BEQZ R3, Try
```

- Many algorithms assume certain operations are atomic (e.g., read/write/etc.). But at some level (possibly in hardware), we must stop relying on atomic operations.
- No concurrent writes, however: a read operation might be concurrent with a write operation or with other read operations.
- Lamport considered three kinds of registers.

Basic Idea

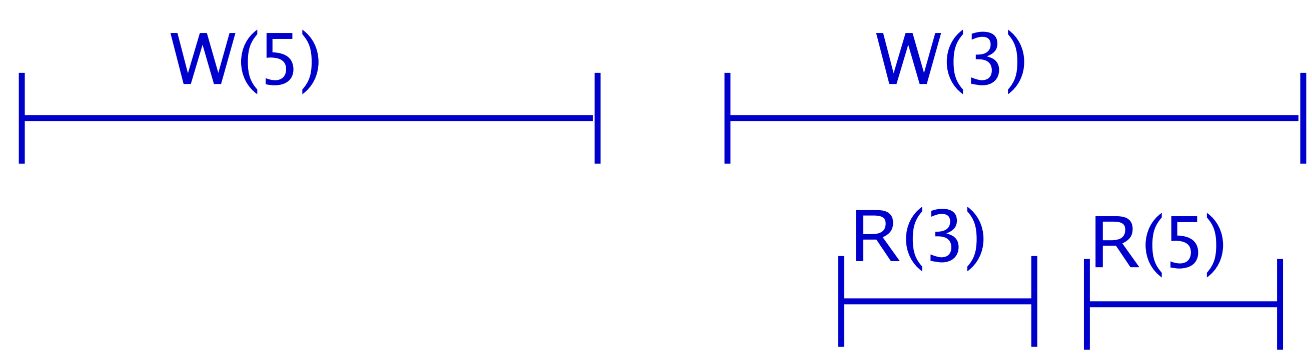
- Interleavings of operation executions gives rise to a partial order over implemented operations.
- In this partial order, A "precedes" B ($A \rightarrow B$) iff the last operation in A is executed before the first in B. Concurrent high-level operations are not ordered by the partial order.



- **Example** $A \rightarrow B$, $B \not\rightarrow C$, $C \not\rightarrow B$.

Types of Registers

- **Safe:** If a read operation is not concurrent with a write operation, then it gets the most recently written value. Otherwise, it can get any value.
- **Regular:** In addition to being safe, also guarantee that, if a read operation is concurrent with a write operation, then the read operation gets either the old value or the new value.
- **Atomic:** There's a total order on operations for any execution. "Time can't go backwards".
- **Example:** Regular but not atomic.



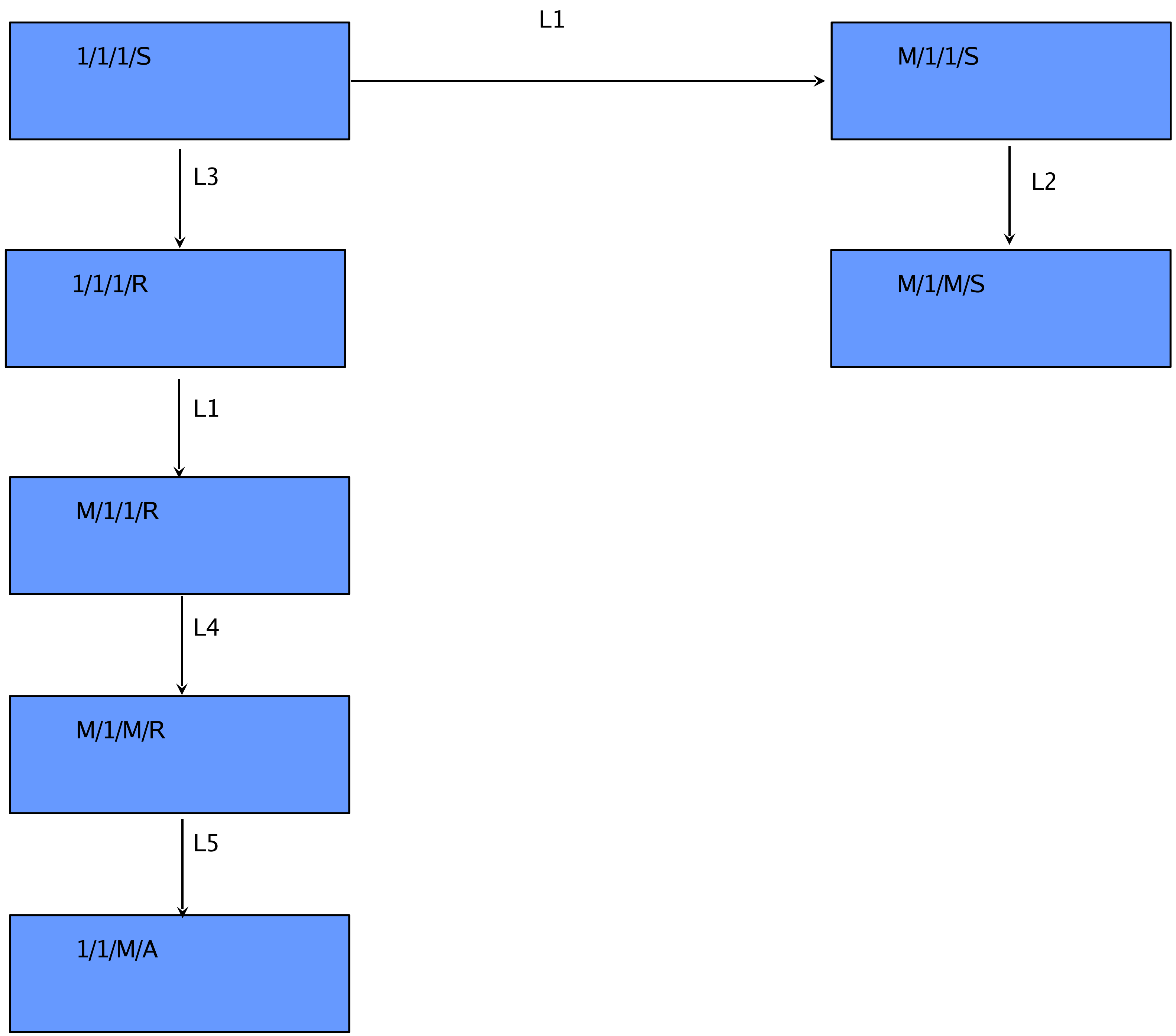
Registers

- Can classify registers by

#readers / #writers / #bits /{safe,regular,atomic}

- Want to implement stronger registers using weaker ones. Weakest is 1/1/1/S. Reasonable hardware assumption.
- We want "wait-free" implementations: each operation must finish in a finite number of steps, regardless of behavior of other processes.

Results



Wait-Free Synchronizatio[Herlihy,`91]

- A *shared object* is a data structure that is shared by a collection of asynchronous processes by means of some fixed set of operations.
- **Examples:** Queue, read/write register, list, etc.
  - Want to answer questions of the form:

"Can object X be used to implement object Y in a wait-free manner?"

- Wait-free means every operation completes in a finite number of its own steps (even if others "fail" by halting).

Consensus

- Herlihy shows that these questions can be answered by focusing on consensus.
  - **Consensus protocol:**
    - Each process has an **input value**.
    - All processes decide on an **output value** by executing a wait-free code fragment. Processes communicate by using shared objects and read/write registers.

Requirements:

- **Consistency**: All non-faulty processes choose the same output value.
- **Validity**: The decision value is the input value of some process.

Herlihy's Hierarchy

- Herlihy has shown that objects can be categorized by "consensus number".
- An object has *consensus number N* iff it can be used to solve consensus for N, but not N+1, processes in a wait-free manner.
- Herlihy has shown that any object X with consensus number N is *universal* in a system of N processes.
  - *Universal* means X can wait-free implement *any* object.

Herlihy's Hierarchy (Cont'd)

Consensus Number	Object
1	**read/write registers**
2	**test&set, swap, fetch&add, queue, stack**
⋮	⋮
2n-2	**n-register assignment**
⋮	⋮
∞	**Memory-to-memory move and swap, CAS, LL/SC**

Informal Arguments

- r/w registers: consensus number =1
  - Similar to 2 generals problem
  - Message reception and modification of state equiv to setting a register
- test&set: consensus number = 2
  - Only 2 distinct values
- LL/SC: consensus number = infinity
  - Any value can be written and read by others

Atomicity

2 meanings:

no other action in between (typ in OS)

either prev state or new state (typ in DB) on failure or visibility of state due to some action

Interrupts most common reason for lack of atomicity on a uniprocessor

Some atomicity (& security!) problems:

Setuid prog (mknod + chown) vs mkdir: on a heavily loaded system:

mkdir foo: mknod foo; (*rm foo; ln /etc/passwd foo;)* chown foo

read/write; pread/pwrite; append to a file (lseek + write)

open with O_CREAT|O_EXCL : (open + creat)

dup2 (fd, fd2) when fd2 open

vs (close (fd2) + fcntl(fd, F_DUPFD, fd2))

int dup(int fd) returns lowest numbered available fd; -1 on error

int dup(int fd, int fd2) returns copy of fd in fd2; closes fd2 if already open; if fd equals fd2, returns fd2 without closing it

pselect: signals & select

Old Unix ways of locking (using link)

```
#define LOCKFILE      "seqno.lock"
#include      <sys/errno.h>
extern int    errno;
my_lock() {
        int    tempfd;
        char   tempfile[30];

        sprintf(tempfile, "LCK%d", getpid());

        /* Create a temporary file, then close it. If the temporary file already exists,
           the creat() will just truncate it to 0-length.*/

        if ( (tempfd = creat(tempfile, 0444)) < 0) err_sys("can't creat temp file");
        close(tempfd);

       / * Now try to rename temporary file to the lock file. This will fail if the lock file
           already exists (i.e., if some other process already has a lock). */

        while (link(tempfile, LOCKFILE) < 0) {
                if (errno != EEXIST) err_sys("link error");
                sleep(1);
        }
        if (unlink(tempfile) < 0) err_sys("unlink error for tempfile");
}
my_unlock() {...  unlink(LOCKFILE) ...}
```

Other “Unix” locking

- Note that creat alone cannot be used
  - Creat does not fail if the file EEXISTs (truncs it)
  - Also cannot check if file exists and then creat it:
  - Race condition: if((fd=open(file, 0))<0) /*race here*/

    fd=creat(file, 0644) /*rw-r—r-- */
- create the lock file, using open() with both O_CREAT (create file if it doesn't exist) and O_EXCL (error if create and file already exists). If this fails, some other process has the lock.
- Try to create a temporary file, with all write permissions turned off. If the temporary file already exists, the creat() will fail. But: does not work if one of the processes is root!
- Other Probs: crashes do not release locks; how long to wait to retry; no notification to waiters; with busy waiting, priority inversion possible

Concurrency & Locking at Various Levels

At HW level: Instruction level

At kernel level

- Between HW events and kernel code
  - Due to Interrupts (interrupt handler and kernel code)
- Between 2 segments of kernel code
  - true concurrency (SMP)
  - interleaved concurrency (2 procs coroutining or two kernel threads)

At thread level inside a process

Across processes on a single machine

Across multiple machines: at HW/kernel/appl level

Interrupts & Kernel code

- Interrupts (or process scheduling) can occur anytime
- Interrupt handler can also call brelse just like ker code
  - However, interrupt handler should not block
  - Otherwise, the process on whose (kernel) stack the interrupt handler runs blocks
- Can expose data structures in an inconsistent state
  - List manipulation requires multiple steps
  - Interrupt can expose intermediate state
- Interrupt handler can manipulate linked lists that kernel code could also be manipulating
  - Need to raise "processor execution level" to mask interrupts (or scheduling)
    - Check & sleep (or test & set) should be atomic

Deadlock and starvation

- Deadlock: two or more processes are waiting indefinitely for an event that can only by caused by a waiting process
  - P0 gets A, needs B
  - P1 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 its suspended
  - May be caused by ordering in queues (priority)

Shared variables

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

Process P_0

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

Process P_1

```
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

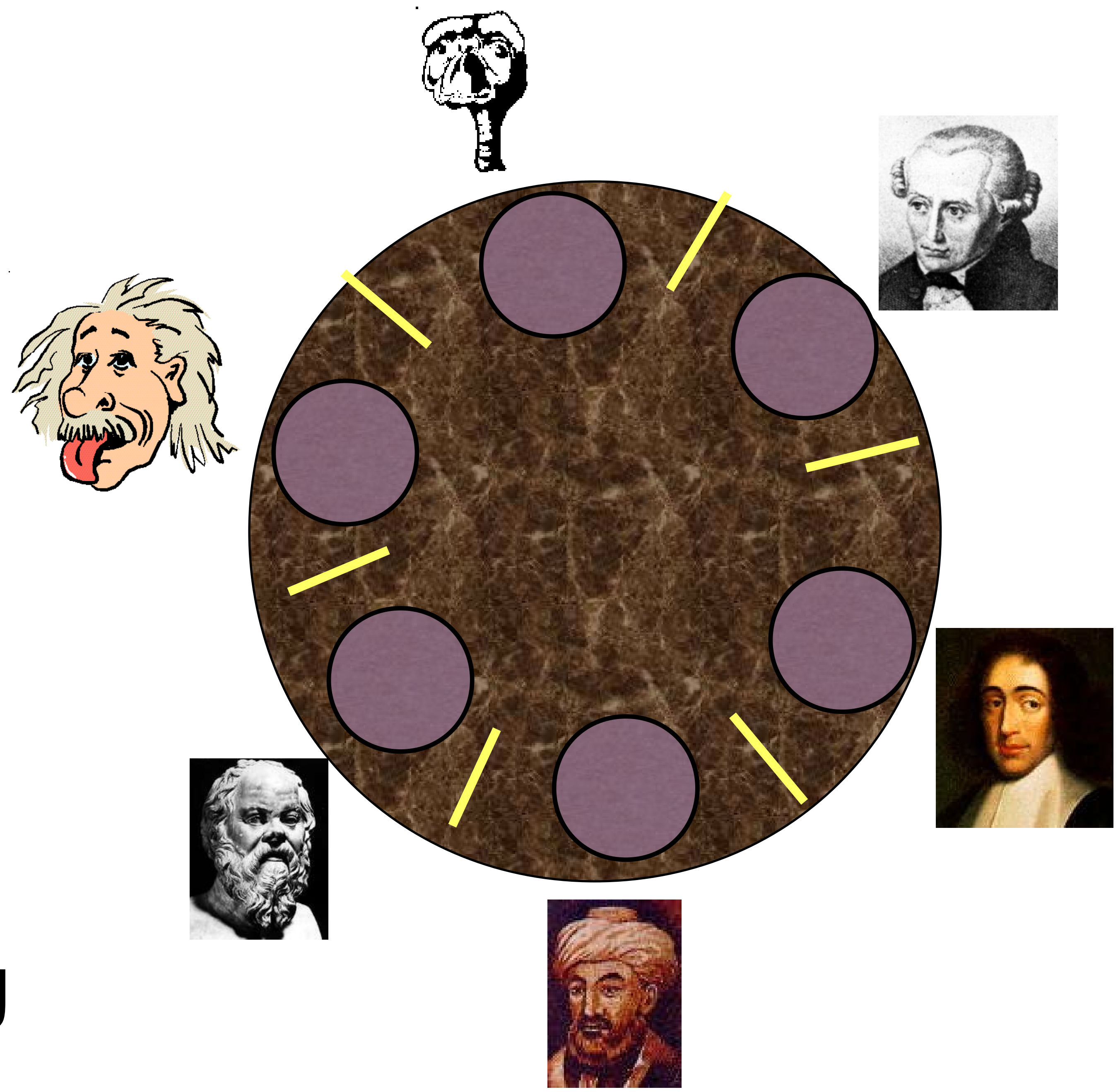
```
…
mutex.down();
nreaders += 1;
if (nreaders == 1) // wait if
  writing.down();  // 1st reader
mutex.up();
// Read some stuff
mutex.down();
nreaders -= 1;
if (nreaders == 0)   // signal if
  writing.up();     // last reader
mutex.up();
…
```

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];
```

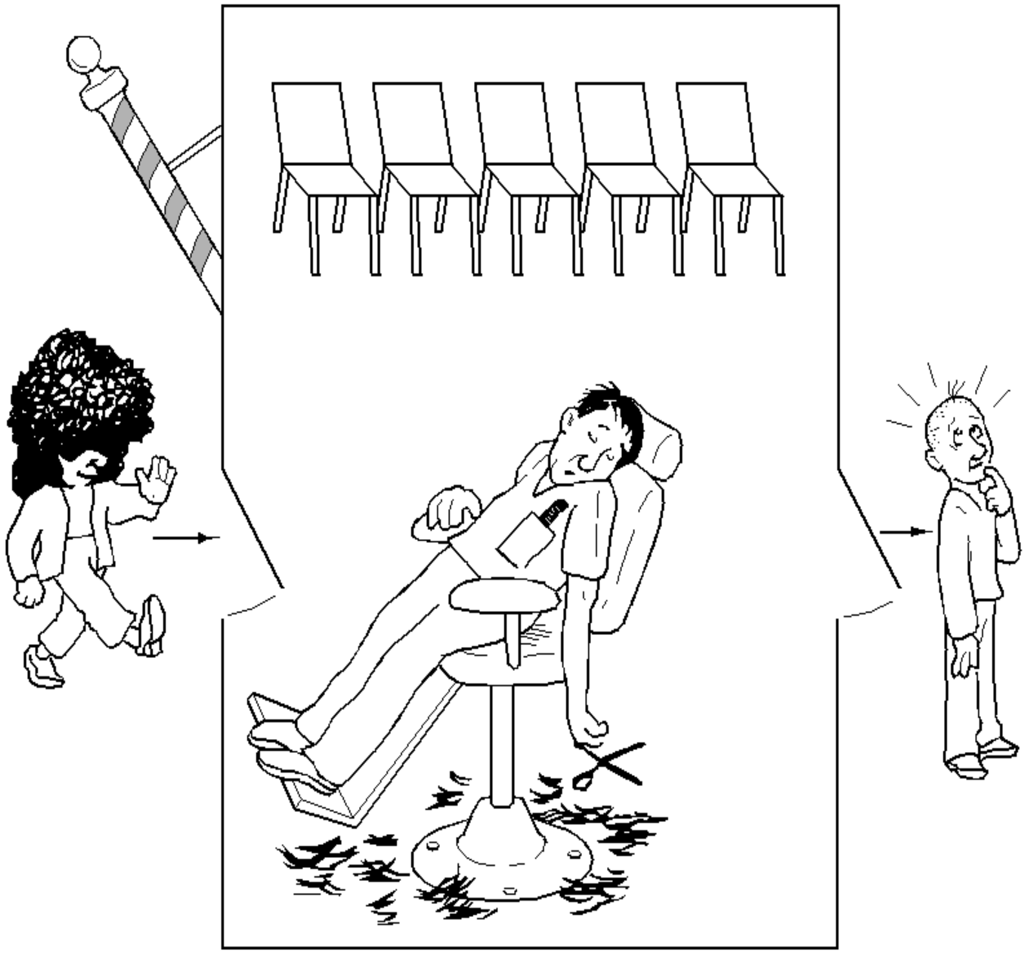
```
void test(int k)
{
 if ((state[(k+n-1)%n)]!=EAT) &&
     (state[k]==HUNGRY) &&
     (state[(k+1)%n]!=EAT)) {
   state[k] = EAT;
   self[k].Signal();
 }
}
```

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 ();
 }
}
```

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?

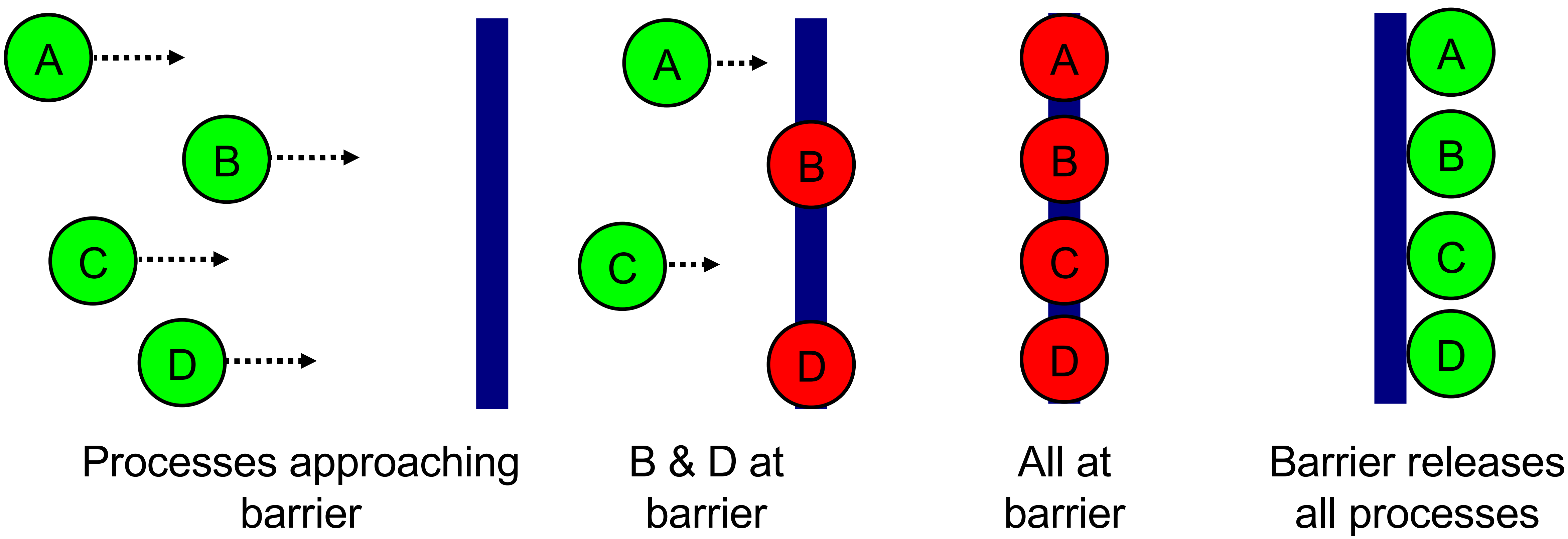
```
Condition::Signal ()
{
  if (semCount > 0) {
    lock->nextCount += 1;
    condSem.up ();
    lock->next.down ();
    lock->nextCount -= 1;
  }
}
```

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