Interprocess Communication and Synchronization

Gonzo: Let's synchronize our watches.
Scooter: We don't have any watches.
Gonzo: That's okay, I don't know what synchronize means anyway.
— *The Muppet Babies*

Baskin Engineering
UC SANTA CRUZ

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:

```
counter += 1;
counter -= 1;
```

Producer

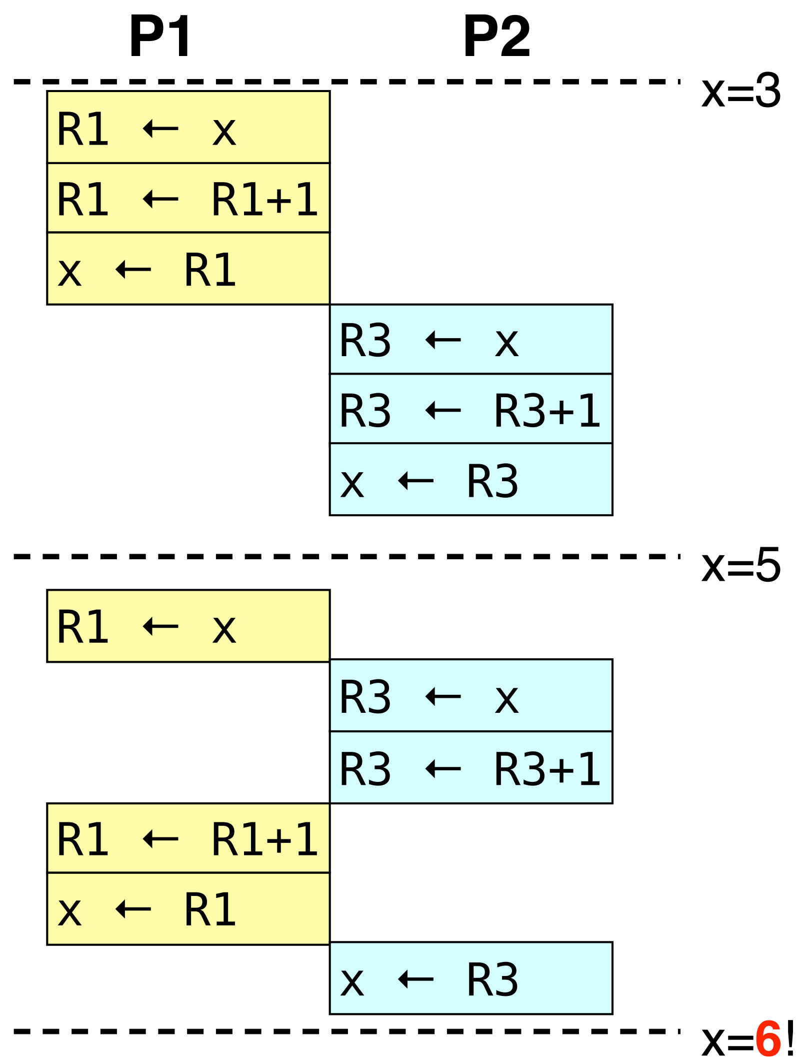
```
Item pitm;
while (1) {
  …
  produce an item into pitm
  …
  while (counter == n)
     ;
  buffer[in] = pitm;
  in = (in+1) % n;
  counter += 1;
}
```

Consumer

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

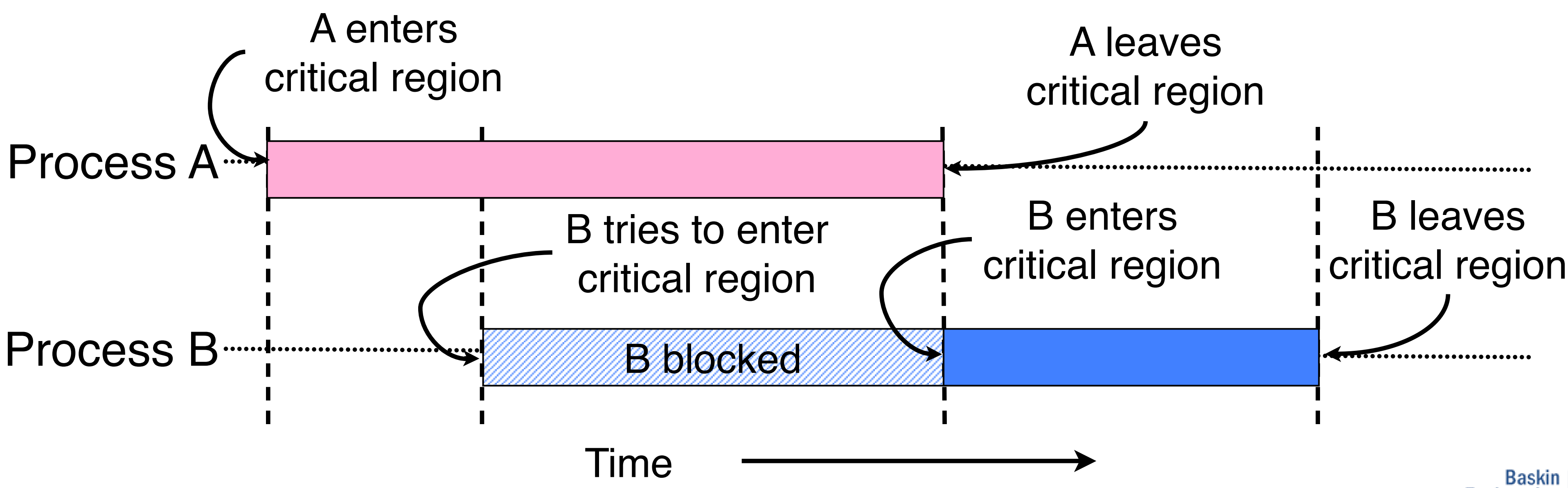
Problem: race conditions

- 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…



Critical regions

- Use critical regions to provide mutual exclusion and help fix race conditions
- Four conditions must hold to provide mutual exclusion
  1. No two processes may simultaneously be in critical region
  2. No assumptions may be made about speeds or number of CPUs
  3. No process running outside its critical region may block another process
  4. 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
  - <<< is lexicographical order on (ticket_number, process ID)
  - (a,b) <<< (c,d) if (a <<< c) or (($a == c$) and ($b < d$))
  - Max $(a_0,a_1,\ldots,a_{n-1})$ is a number k such that $k \geq a_i$ for all i
    - Note: may be *larger* than all of them!
- Shared data
  - `choosing` initialized to 0
  - `number` initialized to 0

```
int n;               // number 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
    - Compare-and-swap supported on x86
  - Turn off interrupts: process won't be switched out unless it asks to be suspended

Mutual exclusion using hardware

- Single shared variable lock
- Two versions
  - Test and set
  - Swap
- Works for any number of processes
- Still requires busy waiting, but code is much simpler
- 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
    - Down(S): **while** ($S \leq 0$) {}; $S \leftarrow S - 1$;
    - Up(S): $S \leftarrow S + 1$;
  - 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 (existing) 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 P_1 only after *A* executes in P_0
  - Use a semaphore initialized to 0
  - Use up() to notify P_1 at the appropriate time
- This is called a *rendezvous*

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
    - Value 1 means the semaphore is available
    - Value 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: better at avoiding 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

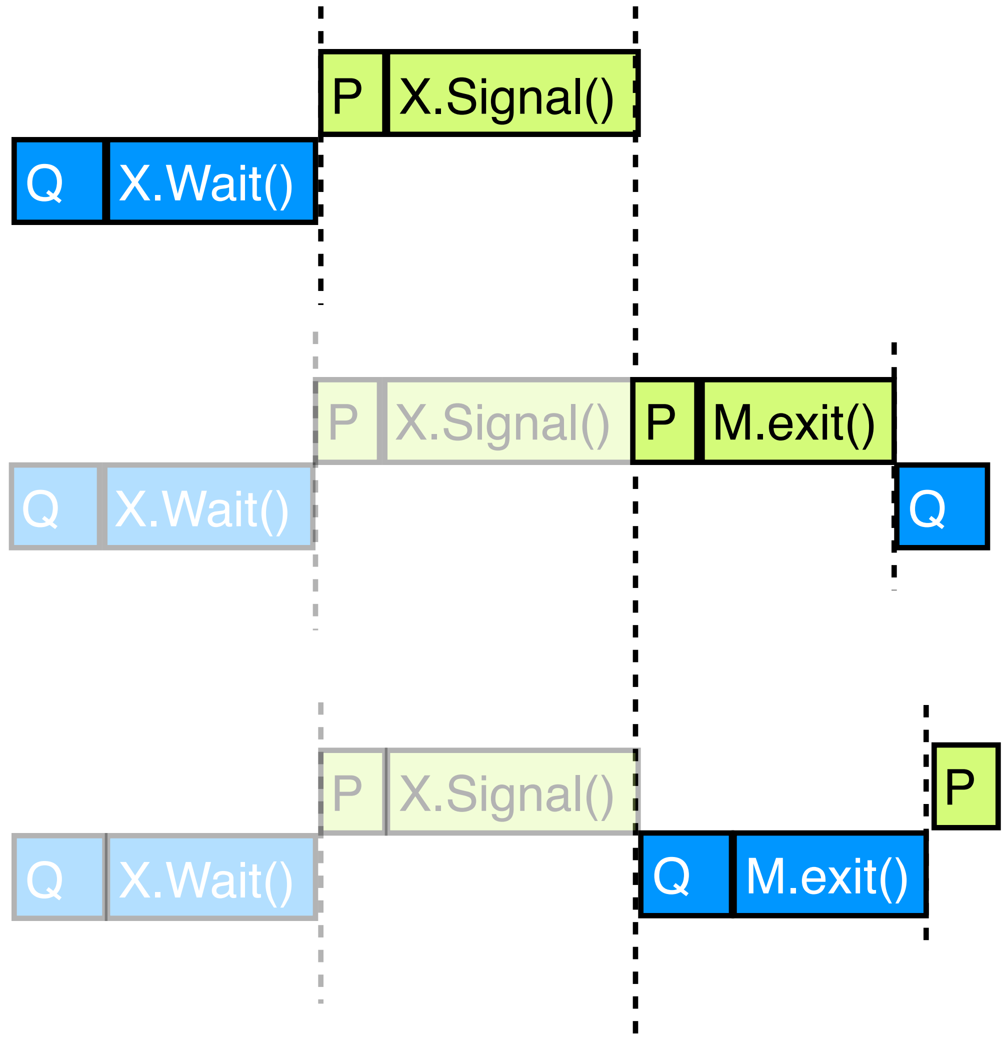
```
monitor mon {
  int foo;
  int bar;
  double arr[100];
  void proc1(…) {
  }
  void proc2(…) {
  }
  void mon() {  // initialization code
  }
}
```

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* in monitor *M*, 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
- Instead, 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 is this necessary?
- *nextCount* indicates whether there are any higher priority waiters

Implementing condition variables

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

- Are these Hoare or Mesa semantics?
- Can there be multiple condition variables for a single Lock?

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

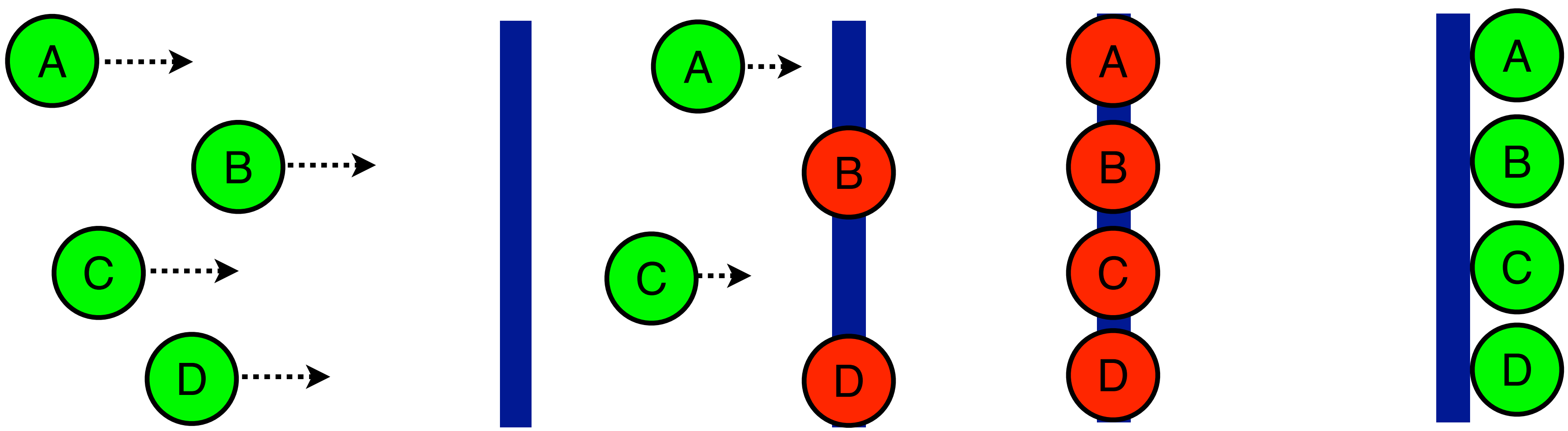
```
Condition::Signal ()
{
  if (semCount > 0) {
    lock->nextCount += 1;
    condSem.up ();
    lock->next.down ();
    lock->nextCount -= 1;
  }
}
```

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



Processes approaching barrier | B & D at barrier | All at barrier | Barrier releases all processes

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 by caused by a waiting process
  - P_0 gets A, needs B
  - P_1 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();
```

Livelock

- Sometimes, processes can still run, but not make progress
- Example: two processes want to use resources A and B
  - P_0 gets A, P_1 gets B
  - Each realizes that a deadlock will occur if they proceed as planned!
  - P_0 drops A, P_1 drops B
  - P_0 gets B, P_1 gets A
  - Same problem as before
  - This can go on for a very long time...
- Real-world example: Ethernet transmission collisions
  - If there's a "collision" on the wire, wait and try again
  - Multiple processes waited the exact same amount of time...

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

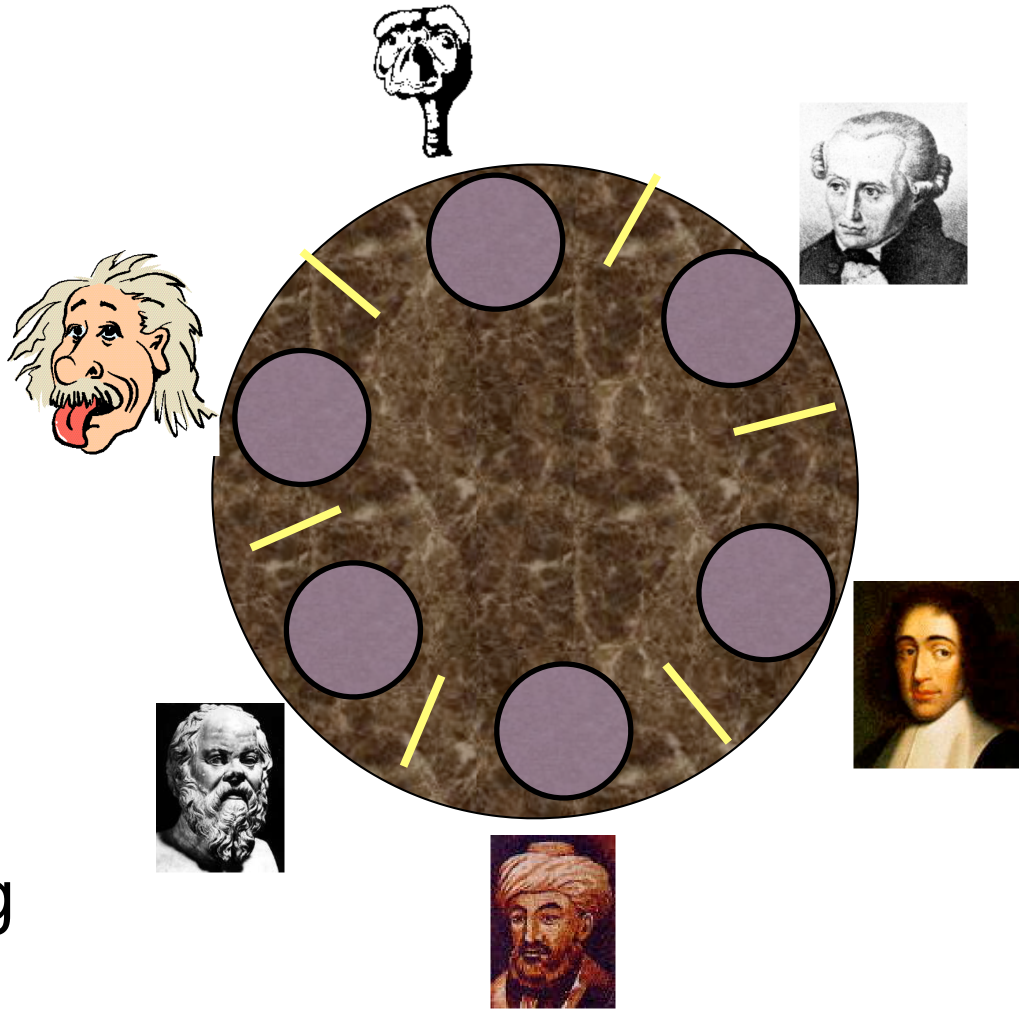
```
…
mutex.down();
nreaders += 1;
if (nreaders == 1) // wait if
  writing.down();  // 1st reader
mutex.up();
// Read some stuff
mutex.down();
nreaders -= 1;
if (nreaders == 0)  // signal if
  writing.up();     // last reader
mutex.up();
…
```

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];
```

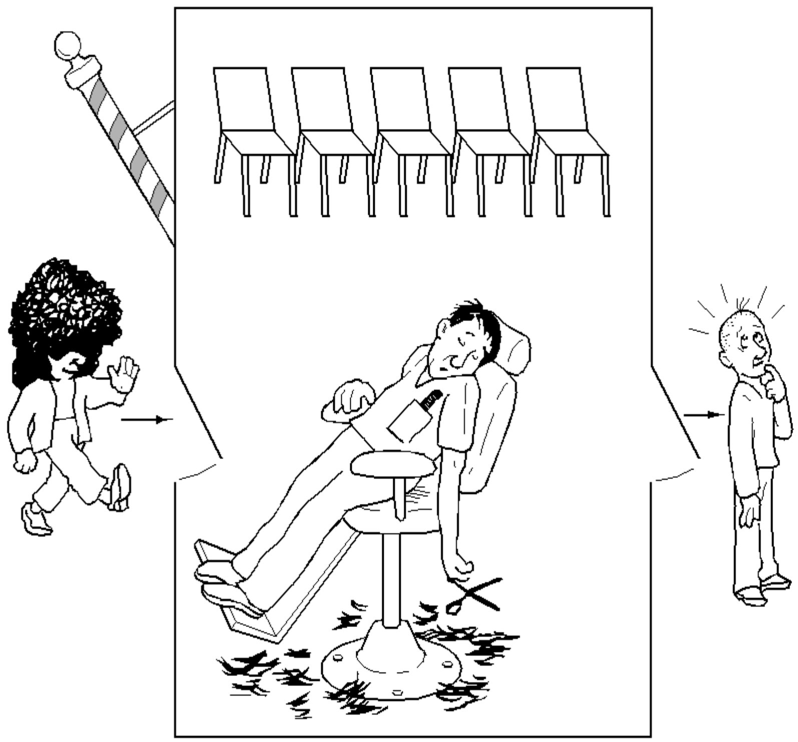
```
void test(int k)
{
  if ((state[(k+n-1)%n)]!=EAT)
&&
      (state[k]==HUNGRY) &&
      (state[(k+1)%n]!=EAT)) {
    state[k] = EAT;
    self[k].Signal();
  }
}
```

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 ();
 }
}
```