While we wait for Brian to figure out the web-page model

www.diku.dk/~vinter/xmp

And we are looking for a larger room!

Threading

Thread packages and thread-programming

Processes and Threads

- Threads are often referred to as lightweight processes
- A thread is simply a process which shares the address space of the process it resides in with the other threads in that process

Thread types

- User level
- Kernel level
- Mixes

Thread Packages

- POSIX Threads
- Solaris Threads
- Win32 Threads
- Java Threads
- + 10^6 custom packages

Types of Threads

- Non-preemptive
- Preemptive
- User level
- Kernel level
- Mixed

User Level Threads

- Non-preemptive switching is fast, Preemptive is slow
- Creating a new thread is fast as is destroying a thread
- Unable to utilize more than one processor

Kernel Level threads

- Preemptive switching is (relatively) fast, Non-preemptive is (relatively) slow
- Creating and destroying threads is slow
- Can utilize more than one processor

Mixed (or both)

- Best of both worlds (BOB)
  - All the advantages of user-level threads combined with MP support
- May introduce a new level of threading

Thread Packages

- Java Threads
- POSIX threads
- Solaris threads
- WIN32 threads

Java Threads

- Integrated into the language

```
class dummyThread extends Thread {
  int id;
  public dummyThread(int id){this.id=id;}
  public void run(){
    System.out.println(“Hello World from thread “+id);
  }
}

dummyThread dt = new dummyThread(42);
dt.start();
dt.join();
```

POSIX Threads

- Language independent library

```
pthread_create(&thread, NULL, worker,(void *)job);
pthread_join(thread);
```

Solaris Threads

- Similar to POSIX however a thread is called a Lightweight process (LWP)
- Introduces a new level of threading on top of LWPs called threads
- LWP are kernel level
- Threads are user level

WIN32 Threads

- API is designed to match the rest of the WIN32 API
- Introduces a second level of threading called fibers
- Threads are kernel level
- Fibers are user-level – and non-preemptive

Thread Control

- Critical regions
- Signal/Wait
- Barriers
- Monitors

Critical regions

- Critical regions are code portions that access data which may be accessed concurrently by another thread
- Unfortunate notation
  - The critical region is really in data
  - But the guards are in code

Critical Region

```
do {
    entry region
      critical region
    leave region
      remainder
} while (1);
```

Mutex mechanism

- The mechanism that performs this check is called a mutex.
- A mutex has two states, that are usually referred to as **unlocked** and **locked**:
  - **unlocked** mutex indicates that the critical region is empty
  - **locked** mutex indicates that there is some thread inside the critical region.

Mutex – how it works

A thread that wishes to access a resource checks the mutex associated with that resource:

- If the mutex is unlocked, it means there is no thread in the critical section:
  - The thread locks the mutex and enters the critical section.
  - When the thread leaves the critical section it should unlock the mutex.
- If the mutex is locked, it means that there is another thread in the critical section:
  - the thread (that is trying to lock the mutex and enter) waits until the mutex becomes unlocked.

Mutex states

- There are two operations defined on a mutex (beside initializing and destroying):
  - **Lock**: checks the state of the mutex
    - locks the mutex if it is unlocked
    - waits until it becomes unlocked.
  - **Unlock**: unlocks the mutex
    - allows any <u>one</u> waiting thread to lock the mutex

Defining and initializing a mutex

A mutex is defined with the type p*thread_mutex_t*, and it needs to be assigned the initial value: *PTHREAD_MUTEX_INITIALIZER*

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

Lock

- A mutex is locked with the function: **pthread_mutex_lock(pthread_mutex_t *mutex)**
- This function gets a pointer to the mutex it is trying to lock.
- The function returns when the mutex is locked, or if an error occurred
  - a locked mutex is not an error, if a mutex is locked the function waits until it is unlocked.

Trylock

- A mutex lock attempt can be made with the function: **pthread_mutex_trylock(pthread_mutex_t *mutex)**
- The function returns true if the mutex is locked
  - false otherwise

Unlocking a mutex

- A mutex is unlocked with the function:

pthread_mutex_unlock(pthread_mutex_t *mutex)

Signal wait

- Used to coordinate progress between threads
- When a thread need another thread to progress before it can continue it will wait
- When the other thread have progressed it will signal the other thread
- Schoolbook example is the producer consumer model

Condition variables

- Address communications between threads that share a mutex
- They are based upon programmer specified conditions

Notifying threads of events

- Problem:
  - Notify another thread that an event has occurred right now (synch) !
  - Thread start waiting until event happens (regardless the past)

Notifying threads - operations

- **wait** - wait <u>until</u> an event occurs.
- **signal** - notify one waiting thread that an even has occurred.
- **broadcast** - notify all waiting threads that an even has occurred.

condition-variable

The pthread library supply a tool for this kind of synchronization.

- The three operations defined on condition-variables are:
  - **wait** - blocks the thread.
  - **signal** - wakes one thread that is waiting on the condition-variable
  - **broadcast** - wakes all threads that are waiting on the condition-variable

How threads wait for a signal

- Just like mutexes every condition variable has a list of threads that are waiting to be signaled
- When a thread calls wait(& c) it adds itself to the waiting list and removes itself from the ready queue
- When signal(& c) is called one thread is extracted from the waiting list and is returned to the ready queue

Note

- The basic operations on conditions are: **signal** and **wait** for the condition
- A condition variable must always be associated with a **mutex**
- **WHY?????**

Note

- What happens when a thread signals on a conditional variable and there is no thread currently waiting?
- A signal is not preserved
  - If one thread signals on a condition variable and no thread is waiting at that moment, the signal "goes away"
  - When a thread waits on the same condition variable it does not catch the previous signal, and has to wait for a new signal

Wait syntax

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

- Atomically unlocks the mutex and waits for the condition variable to be signaled.
- The thread execution is suspended and does not consume any CPU time until the condition variable is signaled.
- The mutex must be locked by the calling thread on entrance to pthread_cond_wait

Signal syntax

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Restarts one of the threads that are waiting on the condition variable
- If no threads are waiting nothing happens.
- If several threads are waiting on exactly one is restarted, but it is not specified which.

Barriers

- Barriers are used to allow a set of threads to ’meet up’
- Only after all threads have called the barrier are they allowed to continue
- Pthreads no longer has a barrier call ☹

Barriers in SMP

- When multiple threads iterates over the same data block we often need to agree when an iteration begins (or ends)
- Barriers comes in three shapes
  - Spinning
  - Blocking
  - Mixed

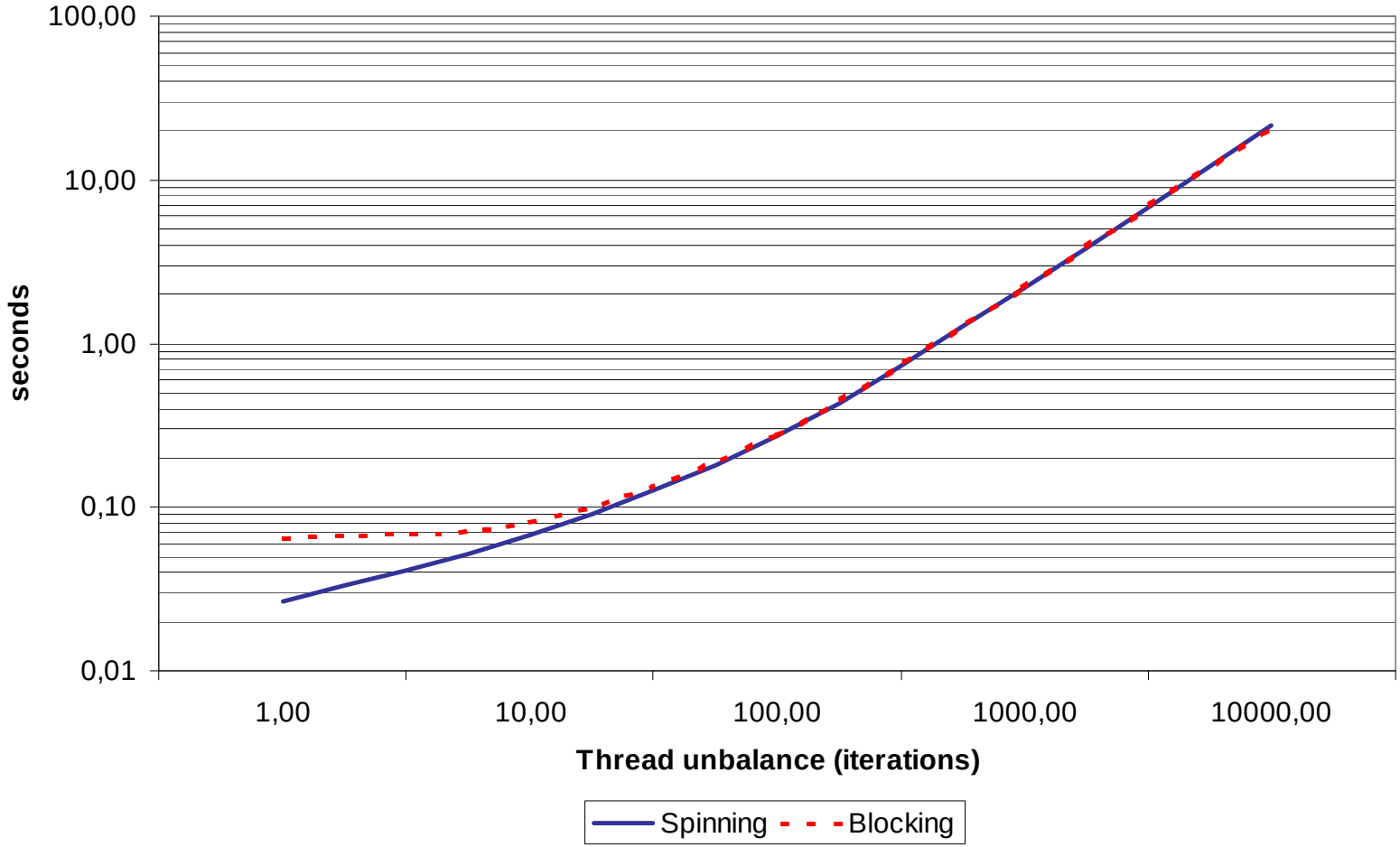
Spinning Barrier

```
class counter {
  int cnt;
  public void counter(){cnt=0;}
  synchronized public int inc(){return ++cnt;}
  synchronized public int reset(){return cnt=0;}
}

class Barrier {
 counter c;
 int members;
 volatile boolean spin;
 public Barrier(int num){members=num; spin=true; c=new counter();}
 public void meet(){
    boolean state=spin;
    if(c.inc()==members){
      c.reset();
      spin=!spin;
    }
    while(spin==state);
 }
}
```

Blocking Barrier

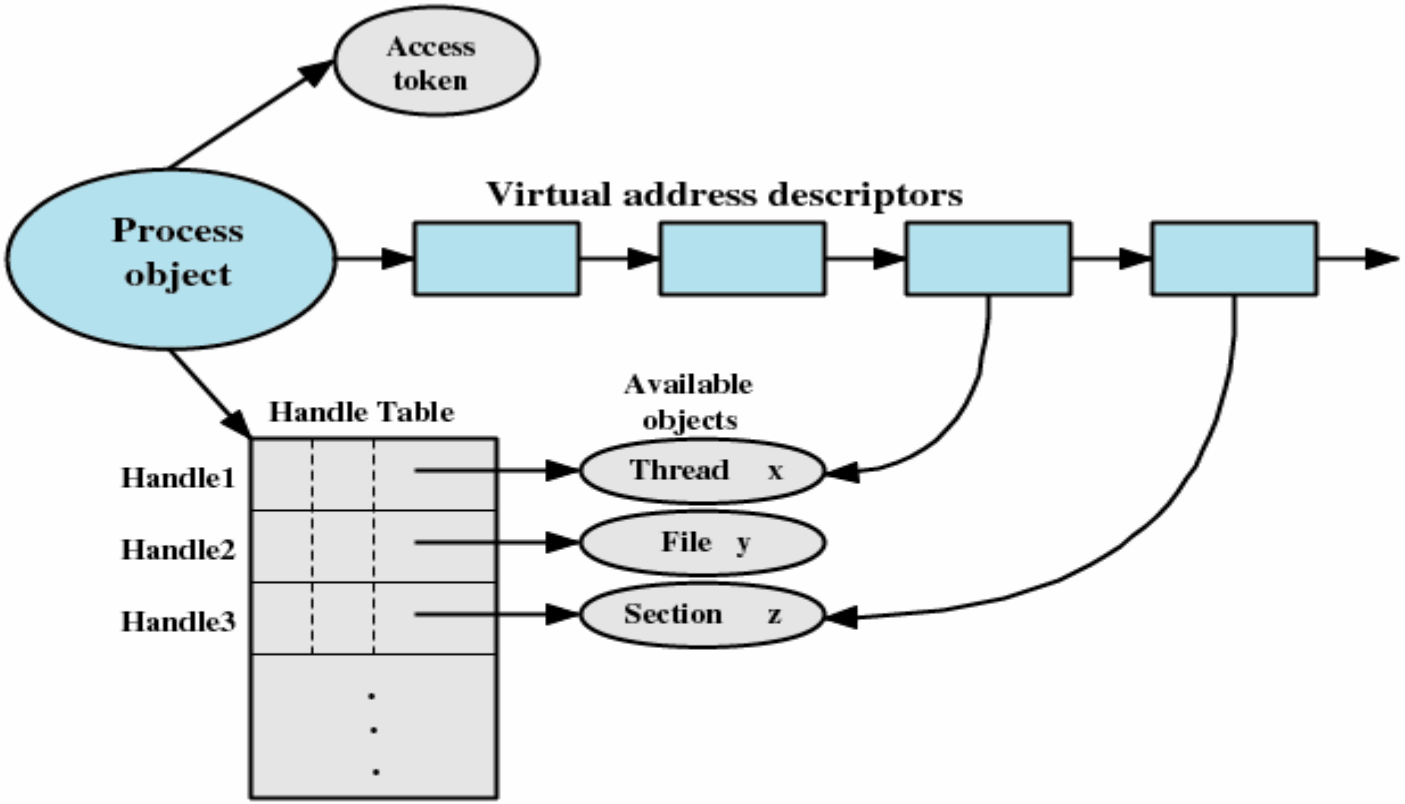
```
class Barrier {
  int cnt, members;
  public Barrier(int members){this.members=members;}
  public synchronized void meet() {
    if(++cnt==members){
      cnt=0;
      notifyAll();
    }
    else try {wait();} catch(Exception e){};
  }
}
```

Spin or block?

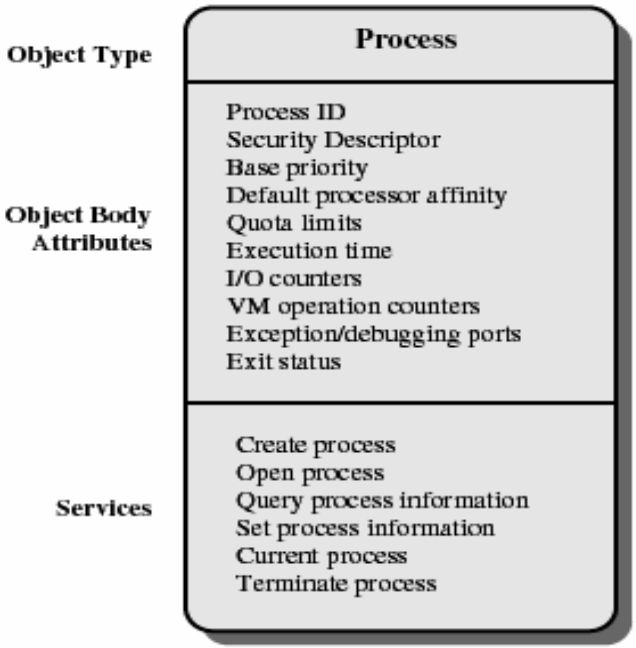


Windows Processes

- Implemented as objects
- An executable process may contain one or more threads
- Both processes and thread objects have built-in synchronization capabilities

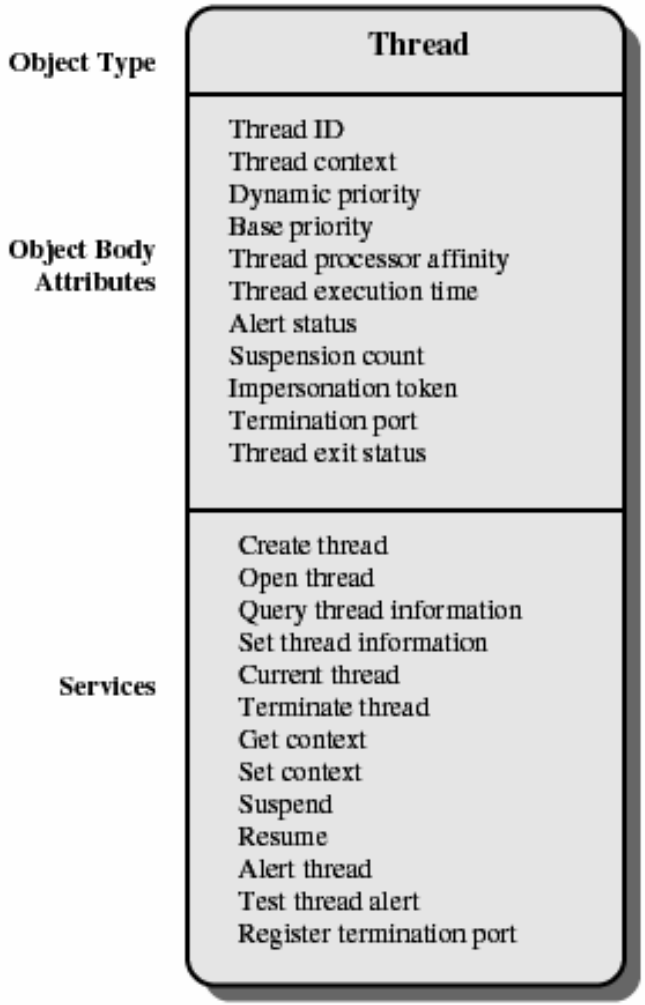

Access token
Process object
Virtual address descriptors
Handle Table
Available objects
Handle1
Handle2
Handle3
Thread x
File y
Section z

Windows Process Object



(a) Process object

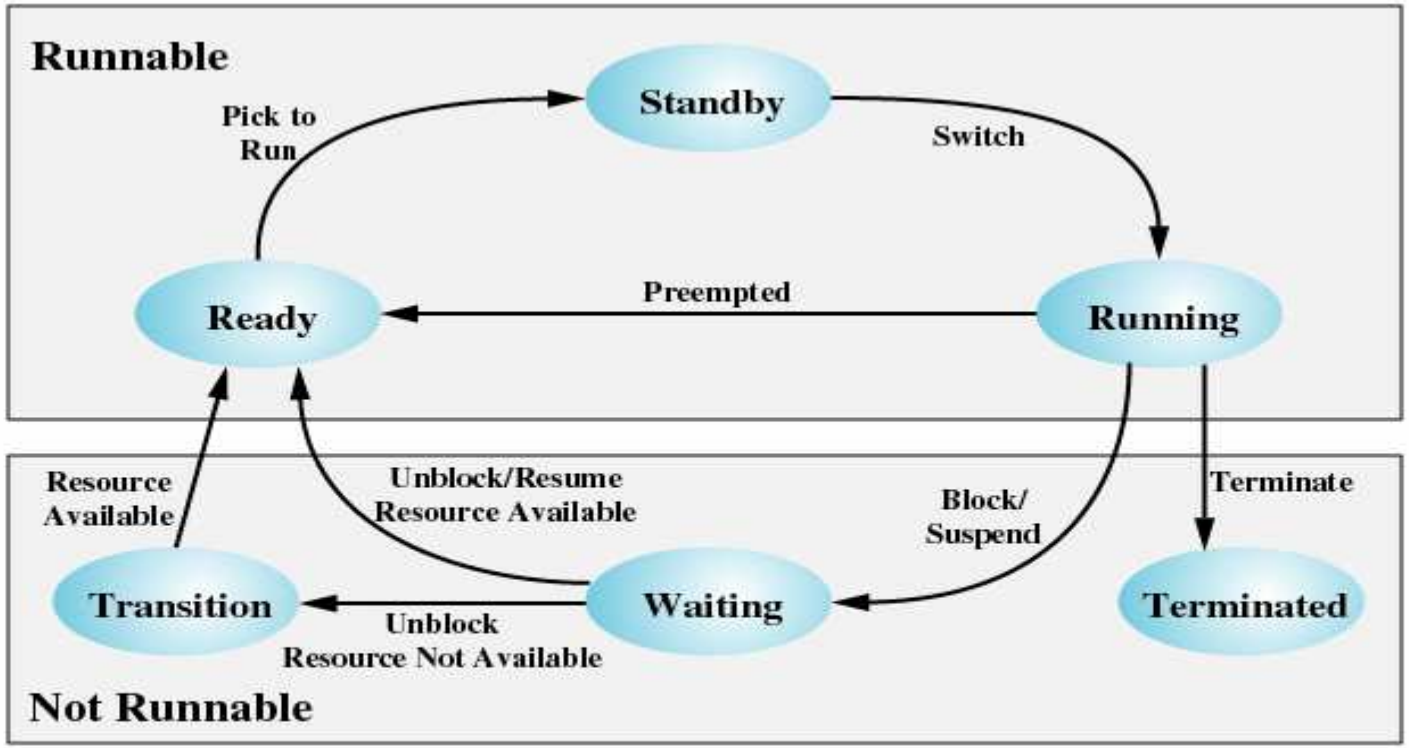
Windows Thread Object



(b) Thread object

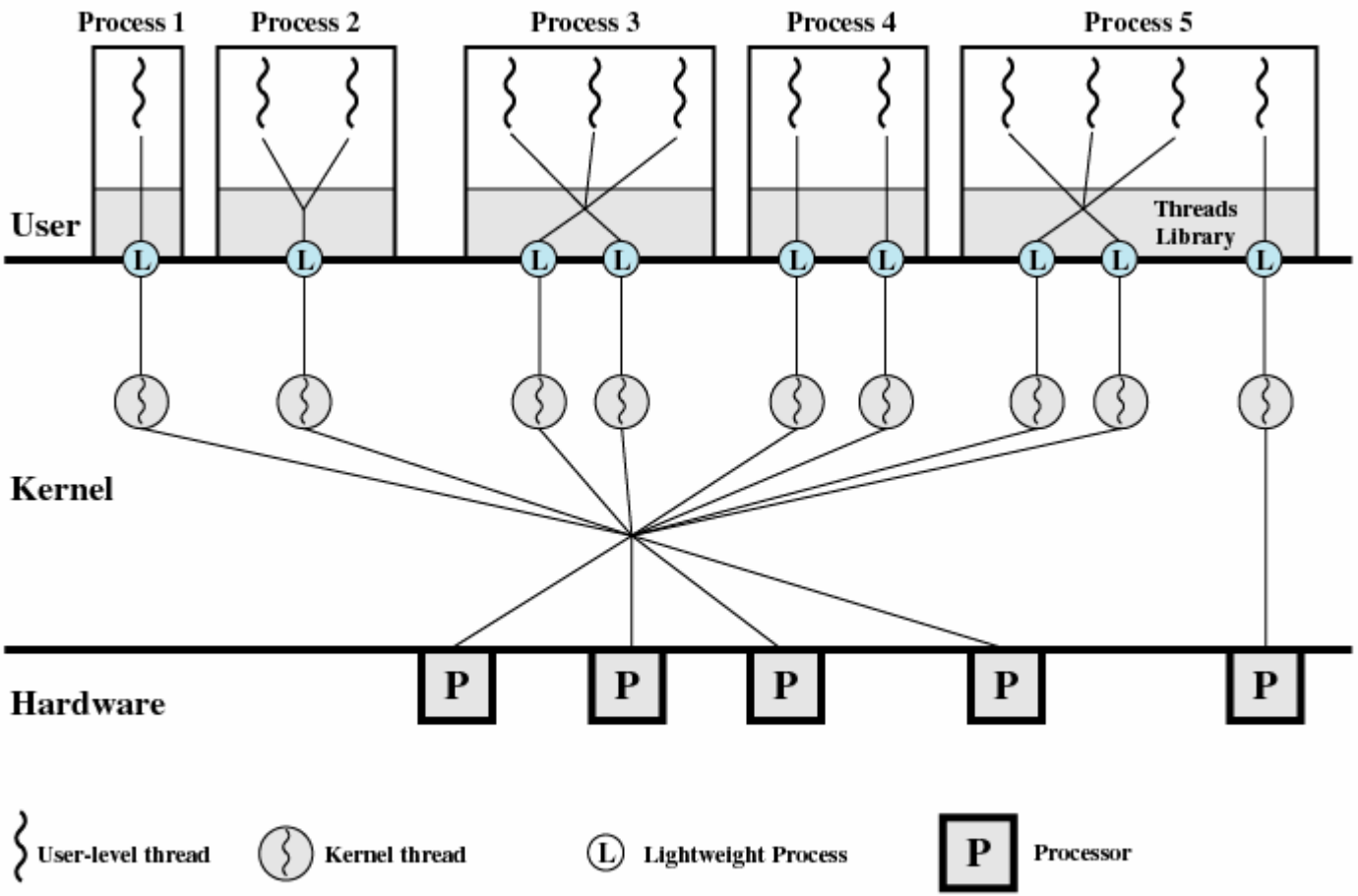
Windows 2000 Thread States

- Ready
- Standby
- Running
- Waiting
- Transition
- Terminated

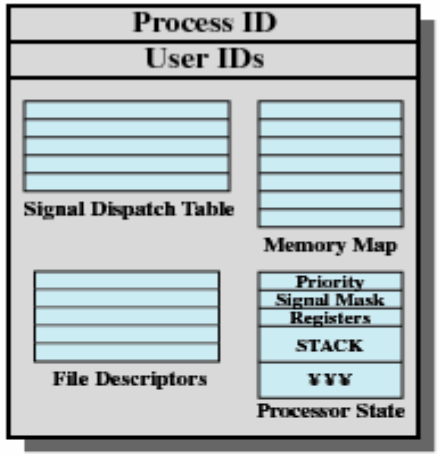

Runnable
Standby
Pick to
Run
Switch
Ready
Preempted
Running
Resource
Available
Unblock/Resume
Resource Available
Block/
Suspend
Terminate
Transition
Waiting
Terminated
Unblock
Resource Not Available
Not Runnable

Solaris

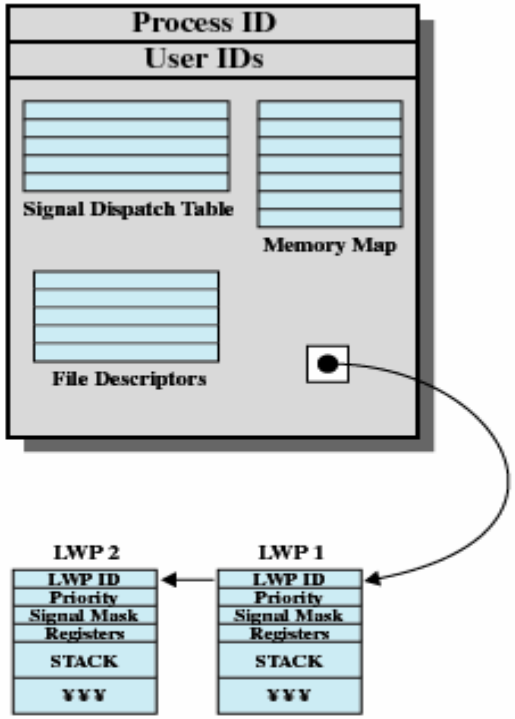
- Process includes the user’s address space, stack, and process control block
- User-level threads
- Lightweight processes (LWP)
- Kernel threads


Process 1
Process 2
Process 3
Process 4
Process 5
User
Threads Library
L
Kernel
Hardware
P
User-level thread
Kernel thread
Lightweight Process
Processor

UNIX Process Structure

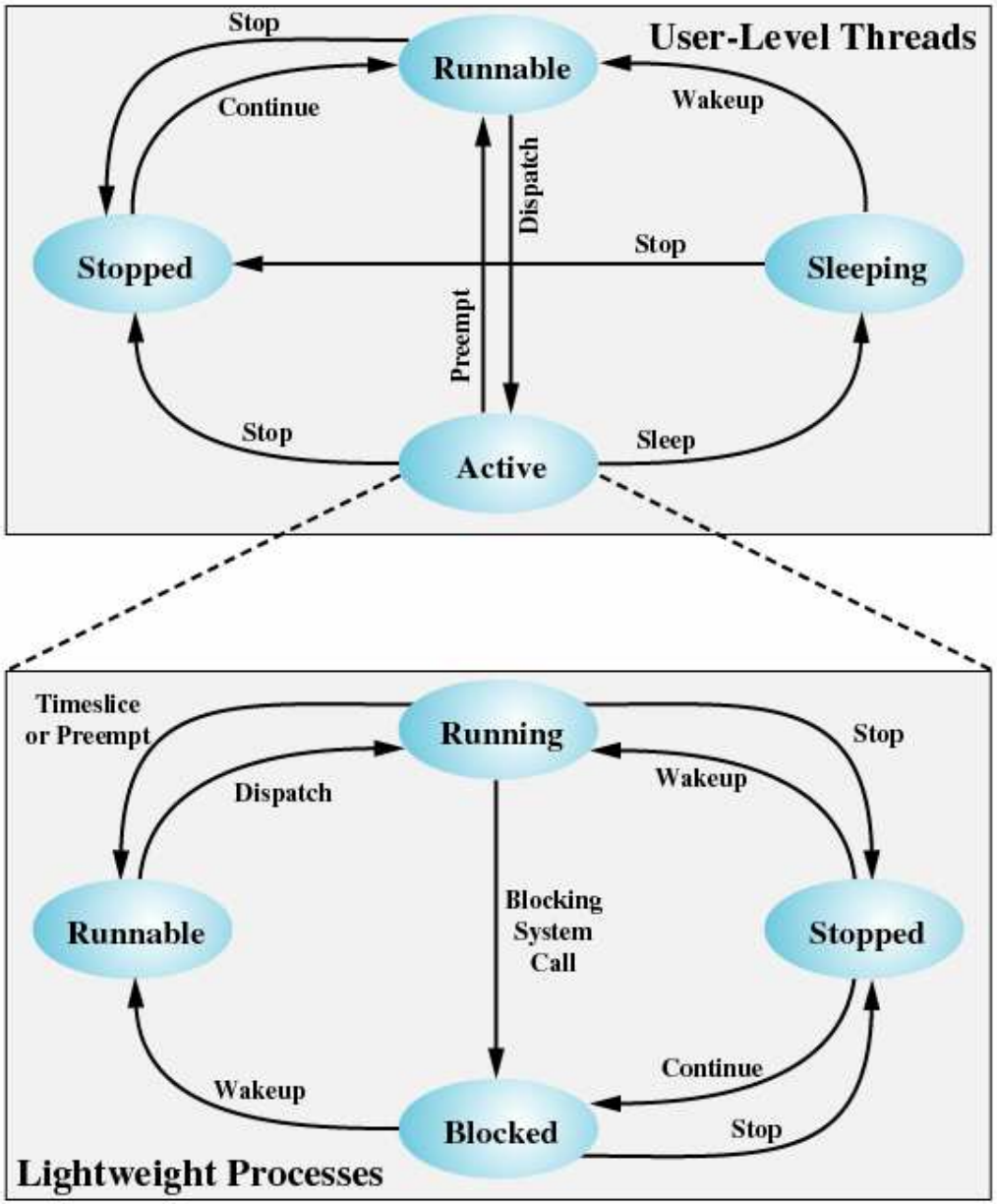


Solaris Process Structure



Solaris Lightweight Data Structure

- Identifier
- Priority
- Signal mask
- Saved values of user-level registers
- Kernel stack
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure

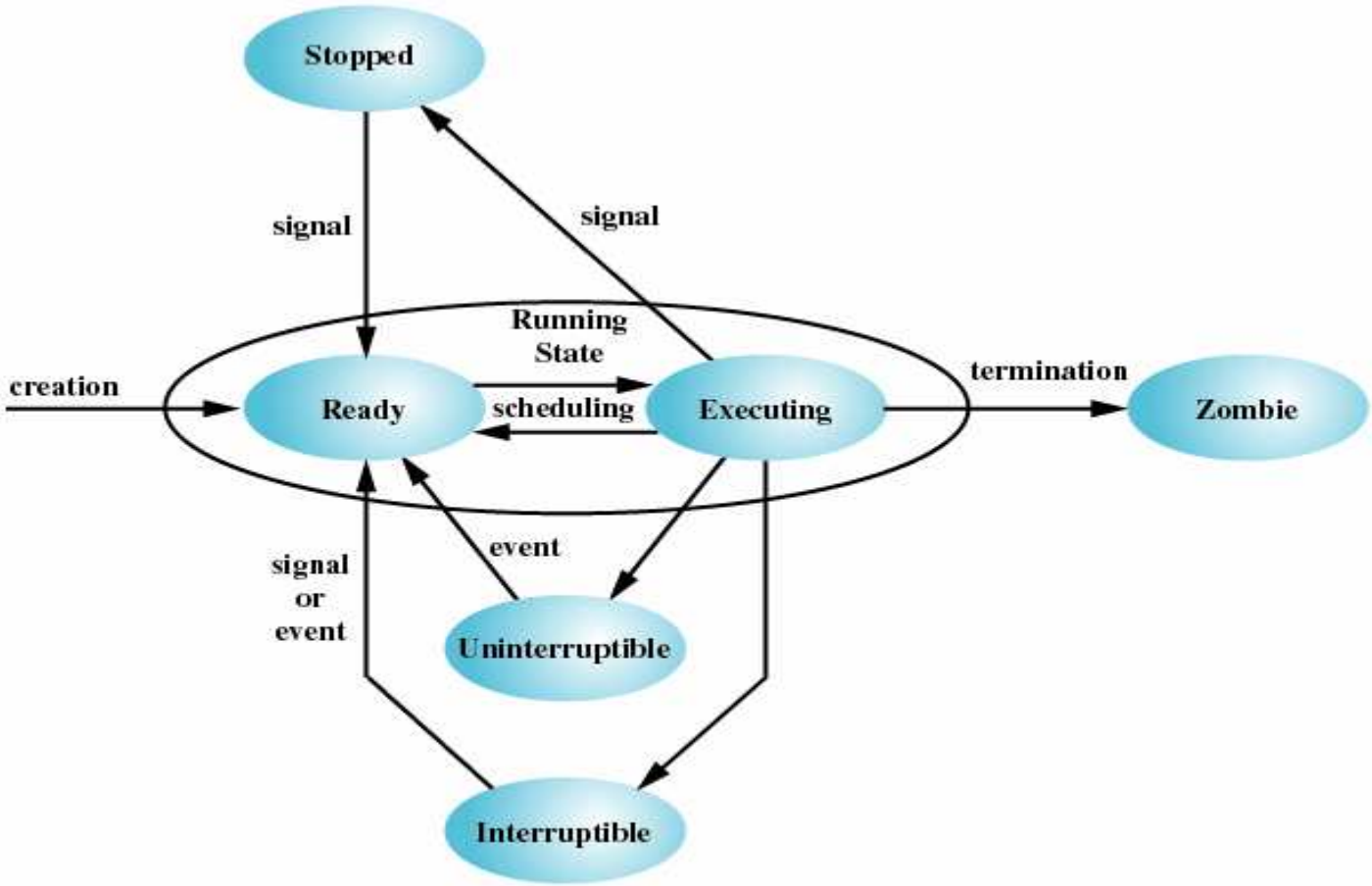

User-Level Threads
Runnable
Stopped
Sleeping
Active
Stop
Continue
Wakeup
Dispatch
Stop
Preempt
Stop
Sleep
Lightweight Processes
Running
Runnable
Stopped
Blocked
Timeslice
or Preempt
Dispatch
Stop
Wakeup
Blocking
System
Call
Wakeup
Continue
Stop

Linux Task Data Structure

- State
- Scheduling information
- Identifiers
- Interprocess communication
- Links
- Times and timers
- File system
- Address space
- Processor-specific context

Linux States of a Process

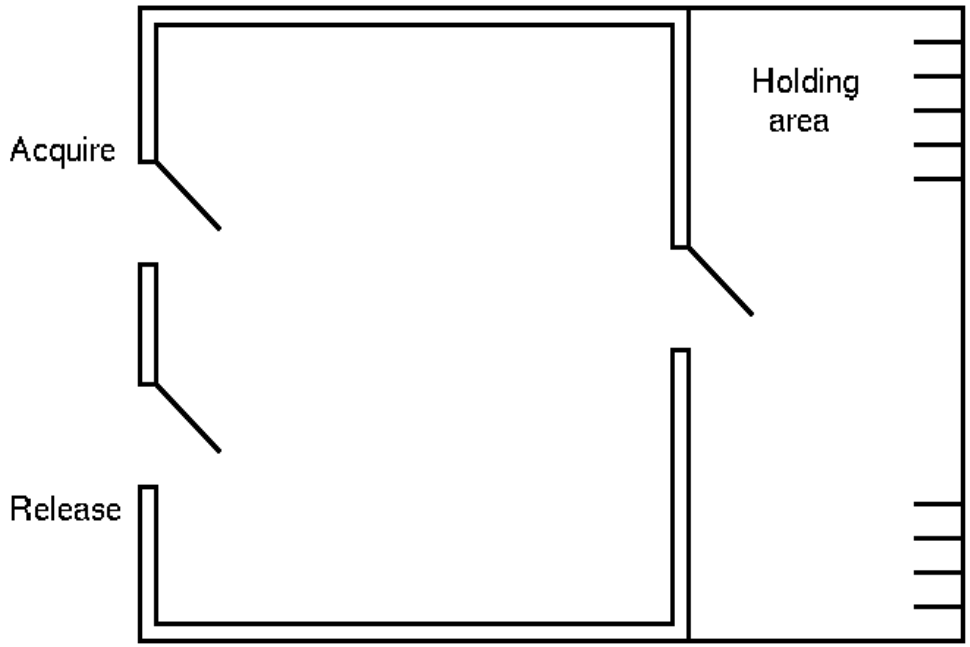
- Running
- Interruptable
- Uninterruptable
- Stopped
- Zombie


Stopped
signal
signal
Running
State
creation
Ready
scheduling
Executing
termination
Zombie
event
signal
or
event
Uninterruptible
Interruptible

Monitors (C.A.R. Hoare)

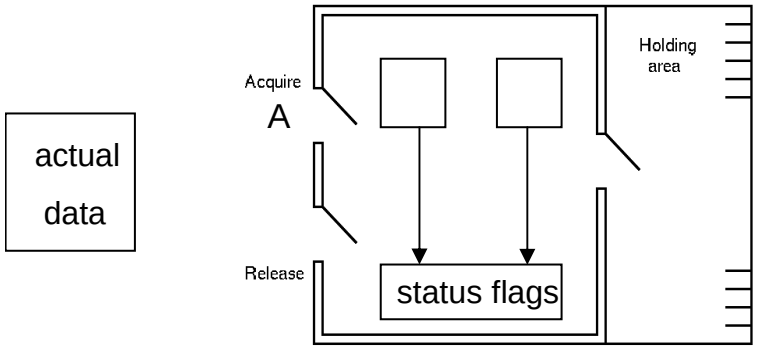
- higher level construct than semaphores
- a package of grouped procedures, variables and data
- processes call procedures within a monitor but cannot access internal data
- can be built into programming languages
- synchronization enforced by the compiler
- only one process allowed within a monitor at one time
- wait and signal operations on condition variables

Blocked processes go into a *Holding Area*

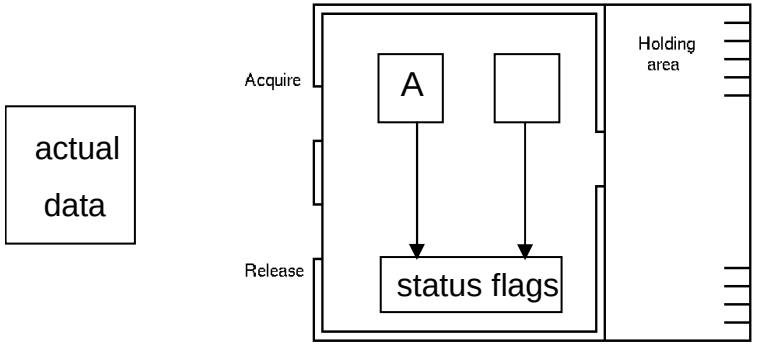


- Possibilities for running signaled and signaling processes
  - let newly signaled process run immediately, and make signaling process wait in holding area
  - let signaling process continue in monitor, and run signaled process when it leaves

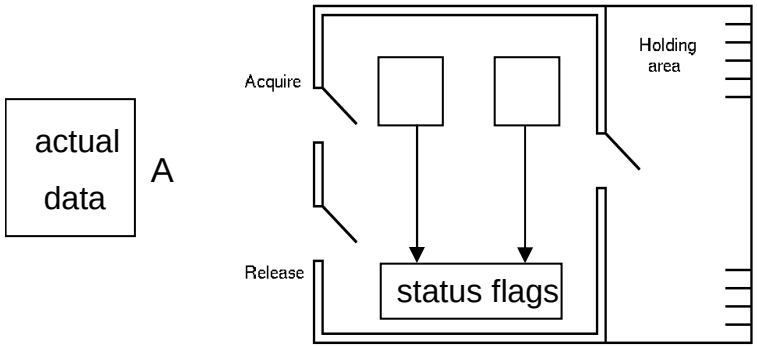
Example: to acquire and release access to some data items/1 :



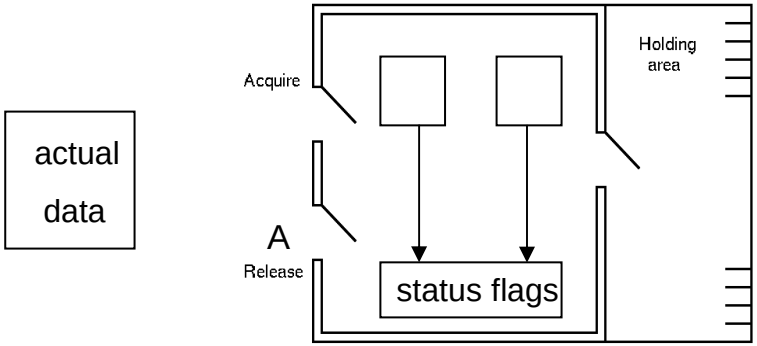
– process A entering monitor to request permission to access data



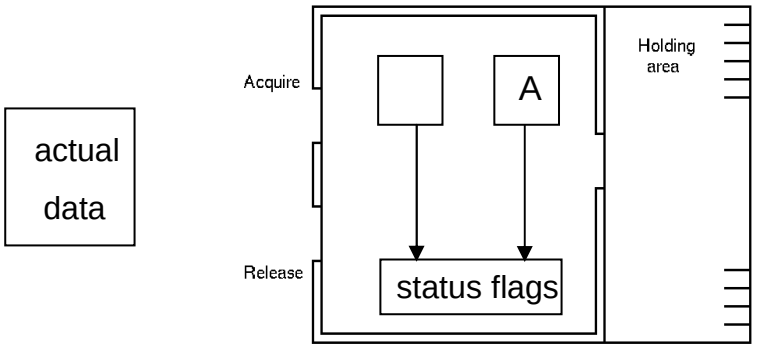
– receiving permission to access data



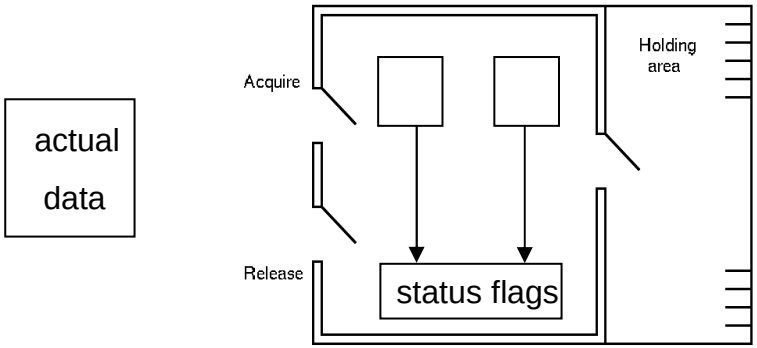
– leaving monitor to access data



– process A entering monitor to release access permission to data

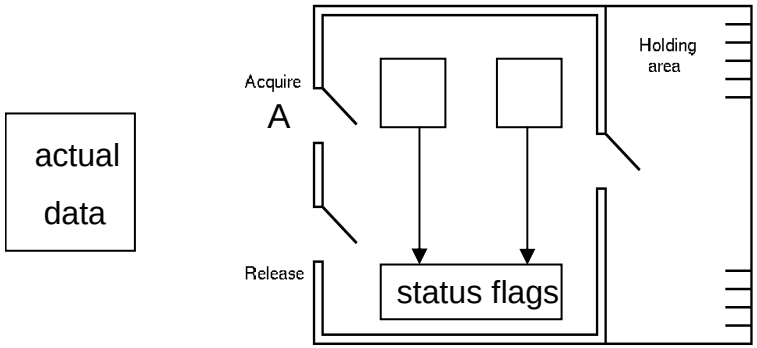


– releasing access permission to data

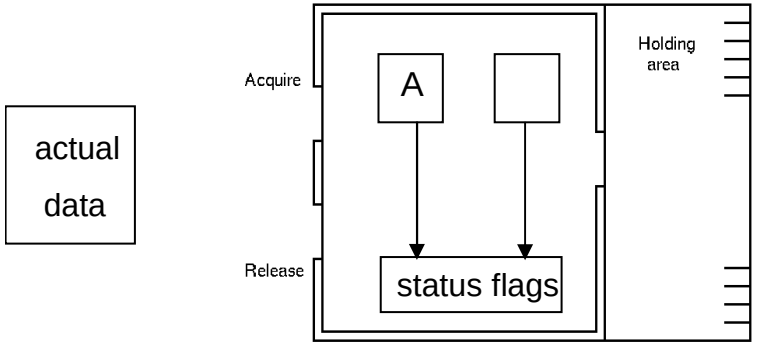


– leaving monitor

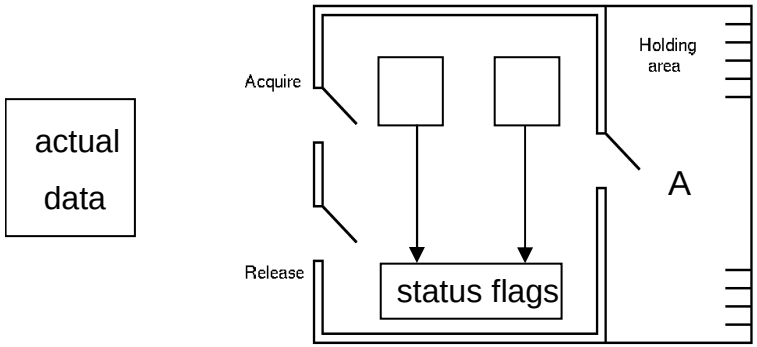
Example: to acquire and release access to some data items/2 :



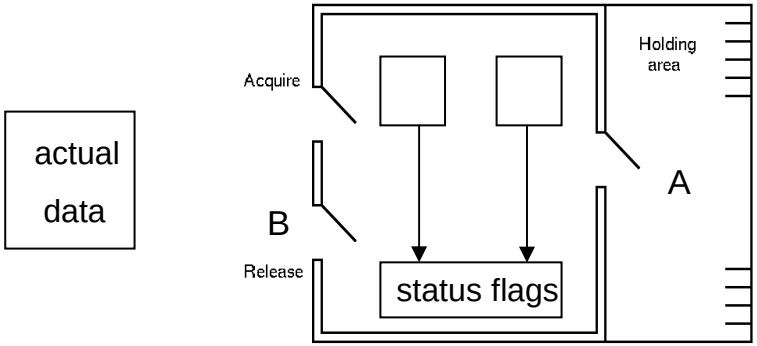
– process A entering monitor to get permission to access to data



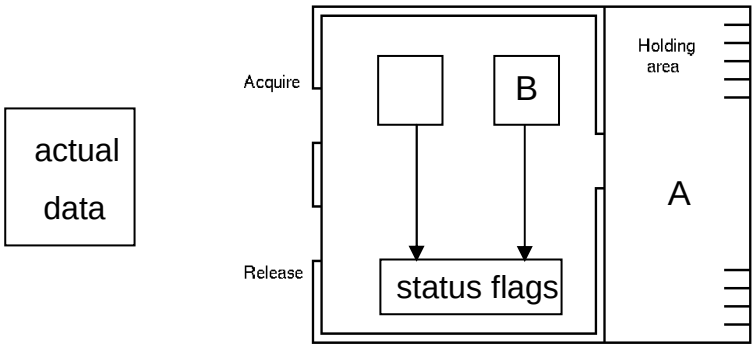
– entering monitor and *not* receiving permission to access data



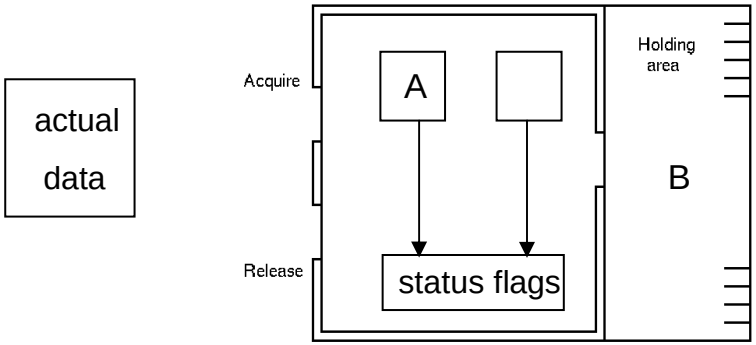
– having to wait in holding area



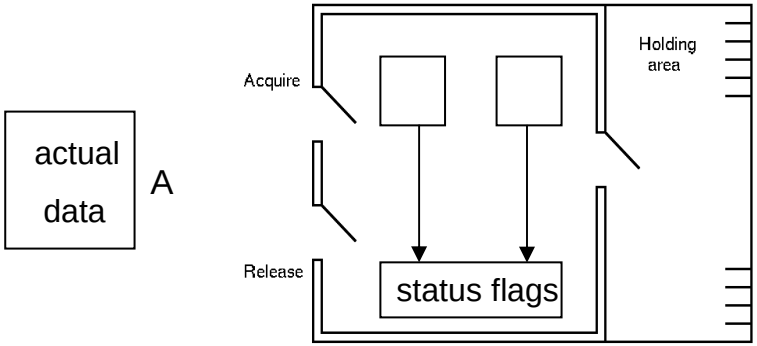
– process B entering monitor to release access to data



– process B releasing access to data

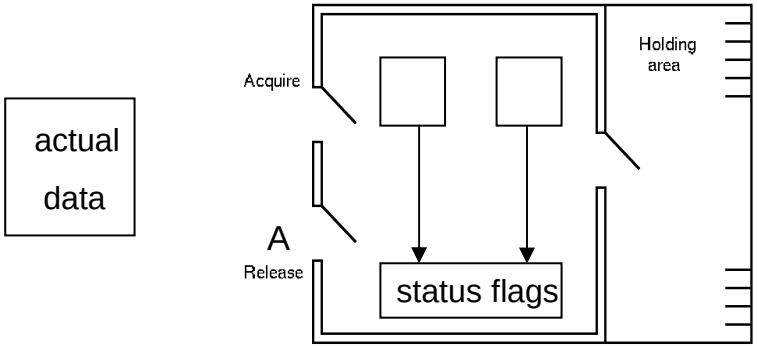


– process B entering holding area whilst process A re-enters monitor to get access permission to data

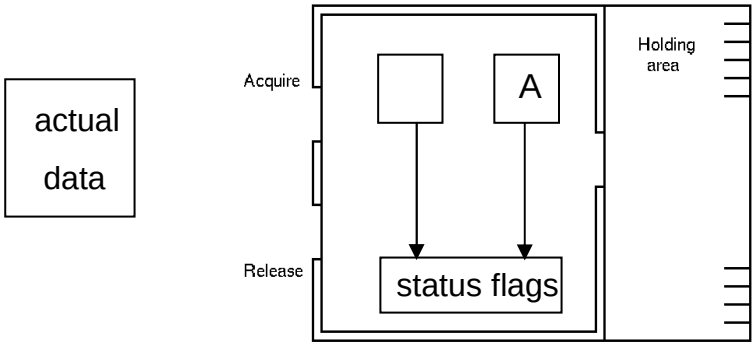


– process A is accessing data

– process B has left holding area and left the monitor



– process A entering monitor to release access to data



– process A releasing access to data

– finally process A leaves monitor

Example: The Dining Philosophers

- five philosophers
  - sometimes they sit and think
  - sometimes they just sit
  - sometimes they want to eat
- one bowl of spaghetti in centre of the table
- five forks
  - one between each place
- need two forks, one from each side, to eat spaghetti

- When a philosopher gets hungry
  - he first gets a fork from his right
  - and then gets a fork from his left
  - he may have to wait for either or both forks if they are in use by a neighbour
- represented by :

```
while (TRUE) {
    P(fork on right);
    P(fork on left);
    eat spaghetti
    V(fork on right);
    V(fork on left);
}
```

- how to avoid possible deadlock?
  - Allow at most four philosophers to eat at once
  - only allow a philosopher to pick up the two forks if they are already available
  - use an asymmetric solution

```
monitor DP {
    condition : self [0:4];
    typedef states = ( thinking, hungry, eating);
    states : state [0:4] = thinking(5);

    entry pickup {
        state [i] = hungry;
        test (i);
        if ( state [i] != eating ) wait ( self [i] );
    }
    entry putdown {
        state [i] = thinking;
        test ( (I+4)%5 );
        test ( (I+1)%5 );
    }
    procedure test (int k) {
        if ( state [ (k+4)%5 ] != eating
         && state [k]==hungry
         && state [ (k+1)%5 ] != eating ) {
            state [k] = eating;
            signal ( self [k] );
        }
    }
}
```

Philosopher process

DP.pickup (i);
eat spaghetti
dp.putdown (i);

– Philosopher can still starve to death!

Implementation of Monitors using semaphores

- For each monitor, there is a *mutex*, initialized to unlocked
- a semaphore *next* , initialized to 0
  - used by *signaling* process to suspend itself
- compiler generates monitor entry and exit code :

```
P(mutex);
monitor code
if (waiting(next) > 0)  V(next);  else  V(mutex);
```

- each condition variable *x* has :
  - a semaphore *x-sem*, initialised to 0

“Monitors” in Java

- Every object of a class that has a *synchronized* method has a “monitor” associated with it
- Any such method is guaranteed by the Java Virtual Machine execution model to execute mutually exclusively from any other synchronized methods for that object
- Access to individual objects such as arrays can also be synchronized
  - also complete class definitions
- Based around use of *threads*
- *One* condition variable per monitor
  - *wait()* releases a lock I.e.enters holding area
  - *notify()* signals a process to be allowed to continue
  - *notifyAll()* allows all waiting processes to continue

- no way to notify a particular thread (but threads can have a *priority*)
- synchronized methods can call other synchronized and non-synchronized methods
- monitor can be *re-entrant* i.e. can be re-entered recursively
- example:

```
class DataBase {
    public synchronized void write ( . . . ) { . . . }
    public synchronized read ( . . . ) { . . . }
    public void getVersion() { . . . }
}
```

  – once a thread enters either of the *read* or *write* methods, JVM ensures that the other is not concurrently entered for the same object
  – *getVersion* could be entered by another thread since not synchronized
  – code could still access a database safely by locking the call rather than by using synchronized methods:

```
DataBase db = new DataBase();
synchronized(db) { db.write( . . . ); }
```

Example:
producer/consumer
methods for a single item :

```
class ProCon {
    private int contents;
    private boolean available = false;

    public synchronized int get() {
        while (available==false) {
            try { wait(); }
            catch (InterruptedException e) { }
        }
        available = false;
        notify();
    return contents;
    }

    public synchronized int put(int value) {
        while (available==true) {
            try { wait(); }
            catch (InterruptedException e) { }
        contents = value;
        available = true;
        notify();
    }
}
```

Java monitor implementation of User-level semaphores

```
class Semaphore {
    private int value;

    Semaphore (int initial) { value = initial; }        // constructor

    synchronized public void P() {
        while (value==0) {
            try { wait(); }
            catch (InterruptedException e) { }
        }
        value = value-1;
    }

    synchronized public void V() {
        value = value+1;
        notify();
    }
}
```

- since the thread calling *notify()* may continue, or another thread execute, and invalidate the condition, it is safer to retest the condition in a *while* loop

```
class BoundedSemaphore {
	private int value, bound;

	Semaphore (int initial, int bound) {		// constructor
		value = initial;
		this.bound = bound;
	}

	synchronized public void P() {
		while (value==0) {
			try { wait(); }
			catch (InterruptedException e) { }
		}
		value = value-1;
		notifyAll();
	}

	synchronized public void V() {
		 while (value==bound) {
			try { wait(); }
			catch (InterruptedException e) { }
		}
		value = value+1;
		notifyAll();
	}
}
```

Java Monitors - CONCERNS

- Threads yield non-determinacy (and, therefore, scheduling sensitivity) straight away ...
- No help provided to guard against race hazards ...
- Overheads too high (> 30 times **???**)
- Learning curve is long …
- Scalability (both in logic and performance) **???**
- Theoretical foundations **???**
  - (deadlock / livelock / starvation analysis **???**)
  - (rules / tools **???**)

“Wot, No Chickens!”

- Peter Welch, University of Kent
- Five Philosophers (consumers)
  - Think
  - Go to Canteen to get Chicken for dinner
  - Repeat
- Chef (producer)
  - produces four chickens at a time and delivers to canteen

“Wot, No Chickens!”

- Philosopher 0 is greedy -- never thinks
- Other philosophers think 3 time units before going to eat
- Chef takes 2 time units to cook four chickens
- Chef takes 3 time units to deliver chickens
  - occupies canteen while delivering

- Simplified code follows -- leaves out exception handling try-catch

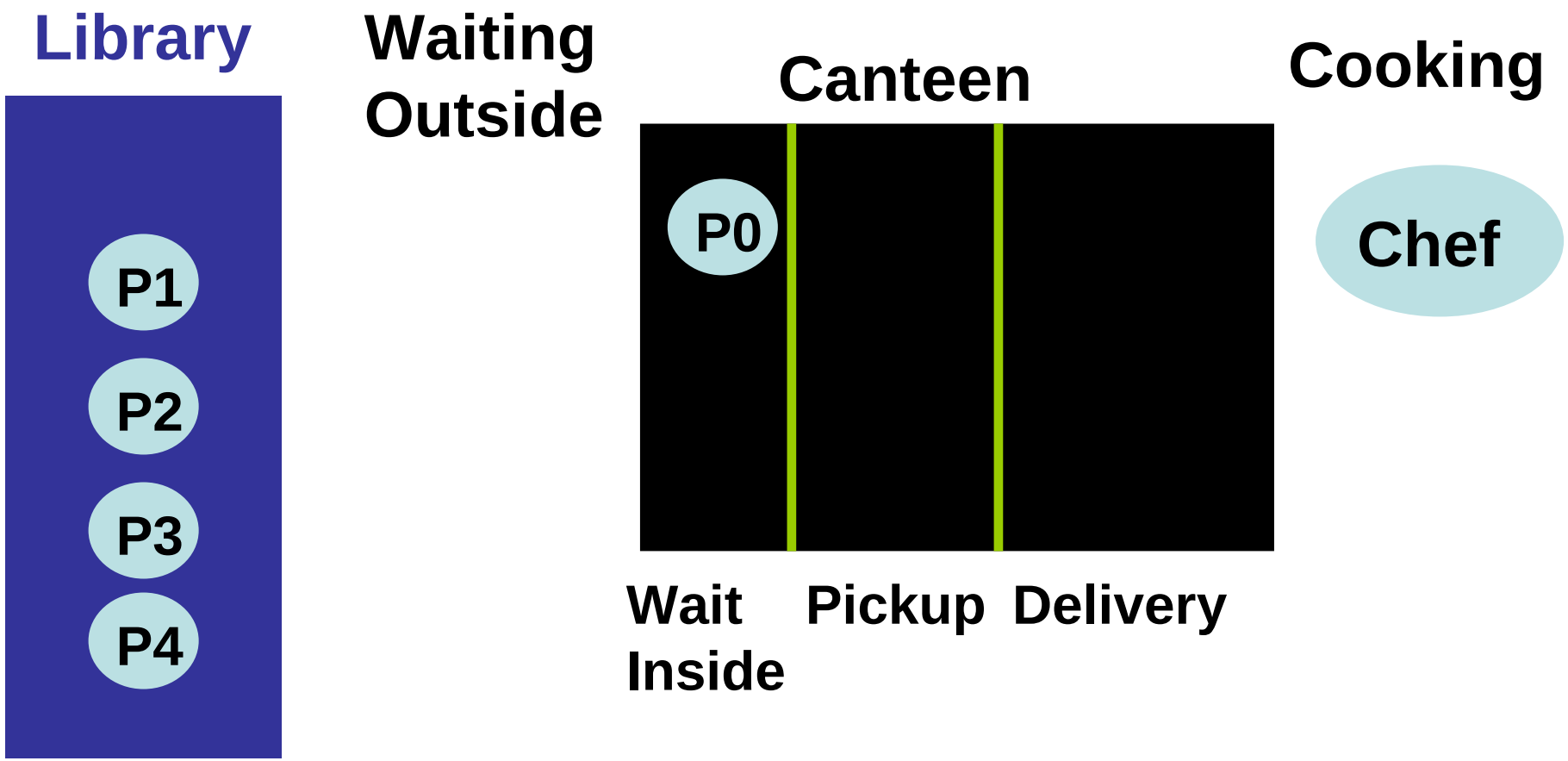
```
class Canteen {

  private int n_chickens = 0;

  public synchronized int get(int id) {
    while (n_chickens == 0) {
      wait(); // Wot, No Chickens!
    }
    n_chickens--;// Those look good...one please
    return 1;
  }

  public synchronized void put(int value) {
    Thread.sleep(3000); // delivering chickens..
    n_chickens += value;
    notifyAll (); // Chickens ready!
} }
```

```
class Chef extends Thread {
  private Canteen canteen;

  public Chef (Canteen canteen) {
    this.canteen = canteen;
    start ();
  }

  public void run () {
    int n_chickens;
    while (true) {
      sleep (2000);//  Cooking...
      n_chickens = 4;
      canteen.put (n_chickens);
    }
  }
}
```

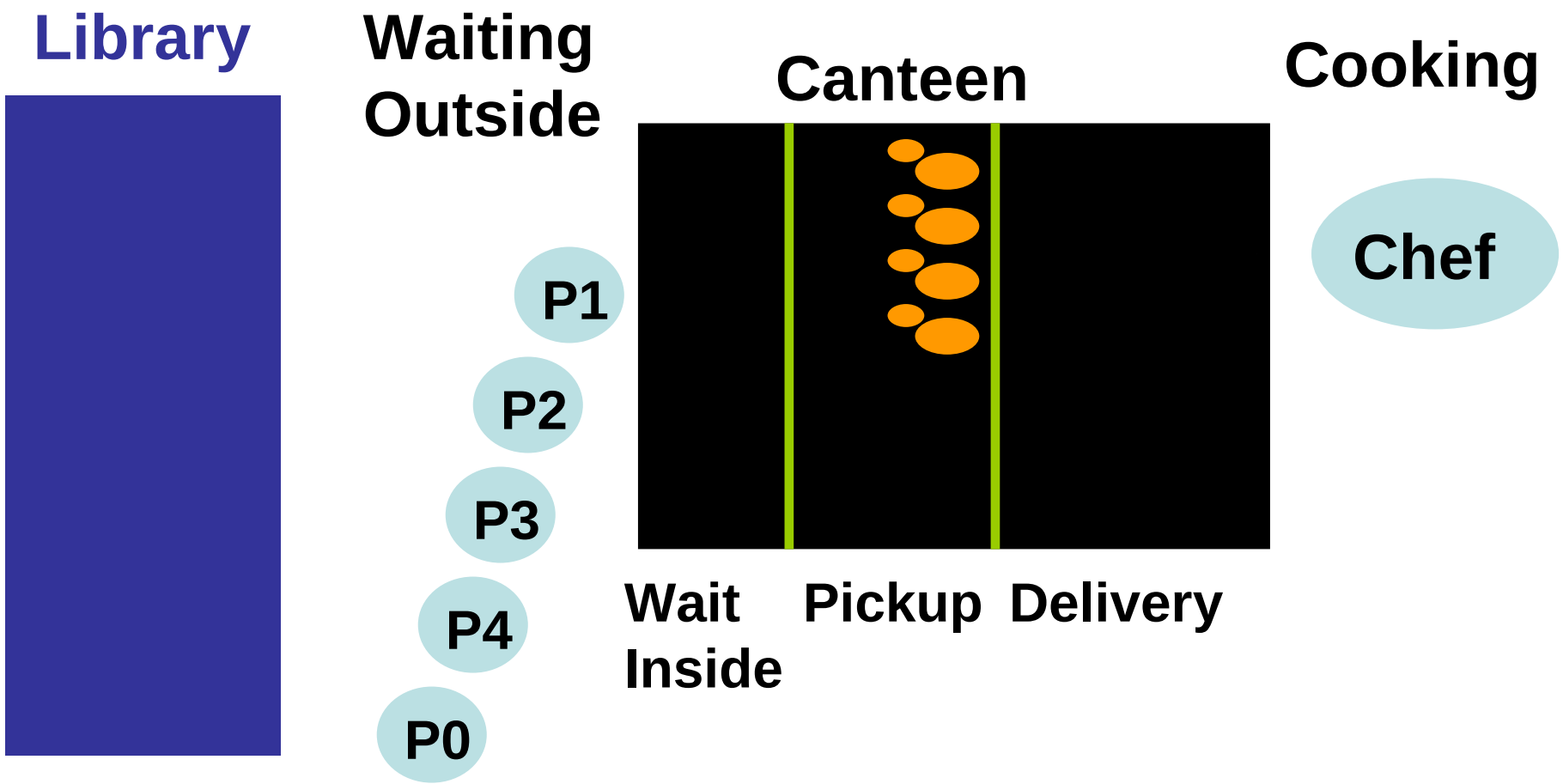
```
class Phil extends Thread {
  private int id;
  private Canteen canteen;

  public Phil(int id, Canteen canteen) {
    this.id = id;
    this.canteen = canteen;
    start ();
  }
  public void run() {
    int chicken;
    while (true) {
      if (id > 0) {
        sleep(3000);            // Thinking...
      }
      chicken = canteen.get(id);// Gotta eat...
    } // mmm...That's good
} }
```

```
class College {

  public static void main (String argv[]) {

    int n_philosophers = 5;
    Canteen canteen = new Canteen ();
    Chef chef = new Chef (canteen);
    Phil[] phil = new Phil[n_philosophers];

    for (int i = 0; i < n_philosophers; i++) {
      phil[i] = new Phil (i, canteen);
    }
  }
}
```

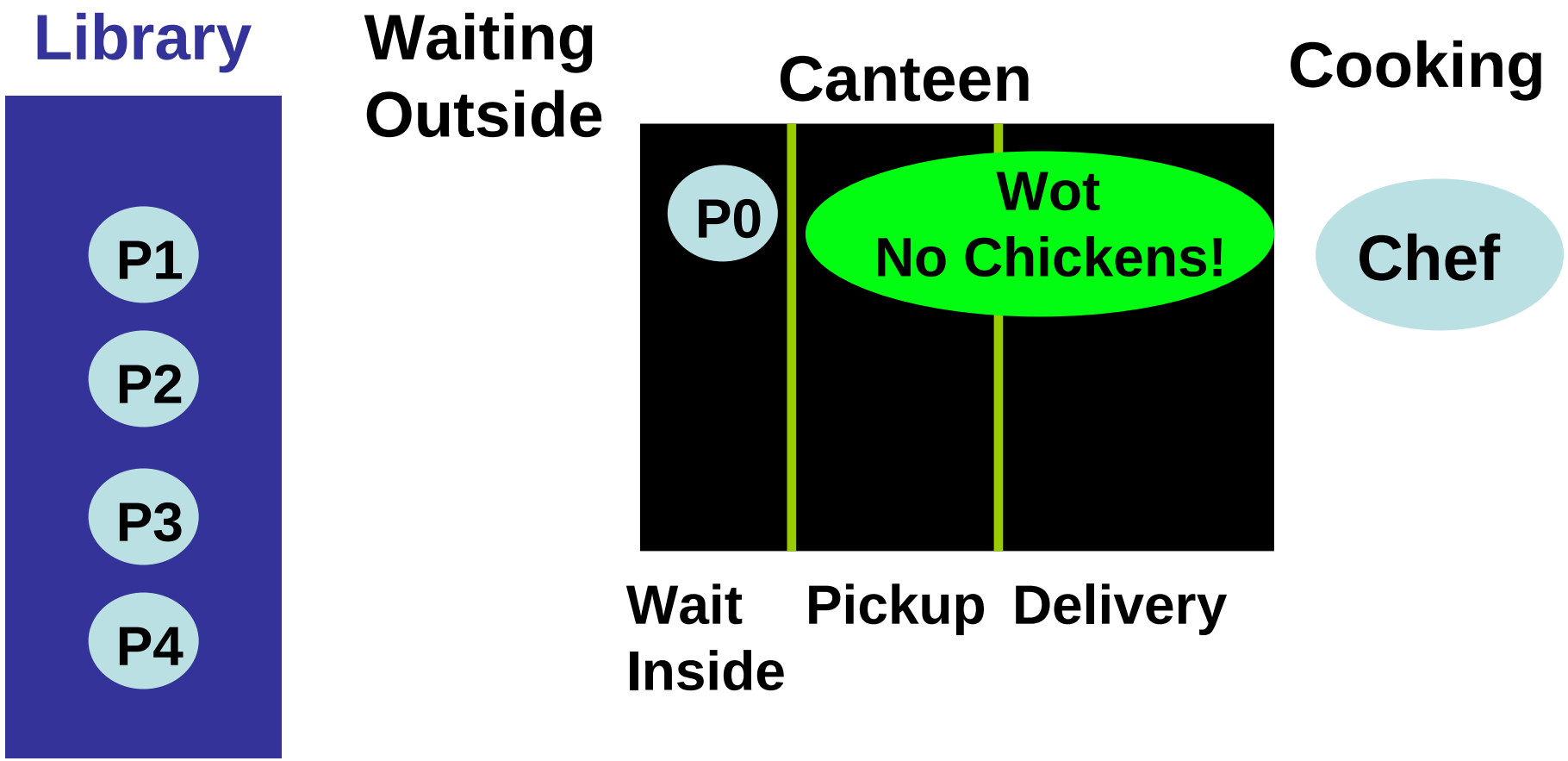
“Wot, No Chickens!”



“Wot, No Chickens!”



“Wot, No Chickens!”



Problems with Java Monitors

- **Semantics of *Notify* / *NotifyAll* ???**
- **Breaks O-O model**
  - Threads are controlled by calling functions inside monitor
  - Depend on notification by other threads
  - Multiple monitors can easily lead to deadlock
  - Monitors can easily lead to starvation
  - <u>Global</u> ! knowledge needed to adequately coordinate multiple monitors