COMP3151/9151 Foundations of Concurrency Lecture 6

Semaphores, Monitors, POSIX Threads, Java

Kai Engelhardt

CSE, UNSW (and data61)

Revision: 1.5 of Date: 2017/08/30 00:52:45 UTC

Semaphores

are our first important programming abstraction for concurrency control.

Ben-Ari suggests to think about a semaphore S as a pair (v, L) of non-negative integers and sets of process descriptor. S should be initialised to some $v > 0$ and $L = \emptyset$.

There are 2 basic operations a process p could performs on S:

wait(S) decrements v if it's positive; otherwise suspends p and adds it to L,

signal(S) unblock a member of L if there is one; otherwise increment v.

Semaphores classes

Semaphores as defined above are known as *weak* semaphores in the literature. If L is a FIFO queue instead of a set, we get a *strong* semaphore.

We often treat sempahores as if they were just an integer with an access policy requiring us to use only the two operations:

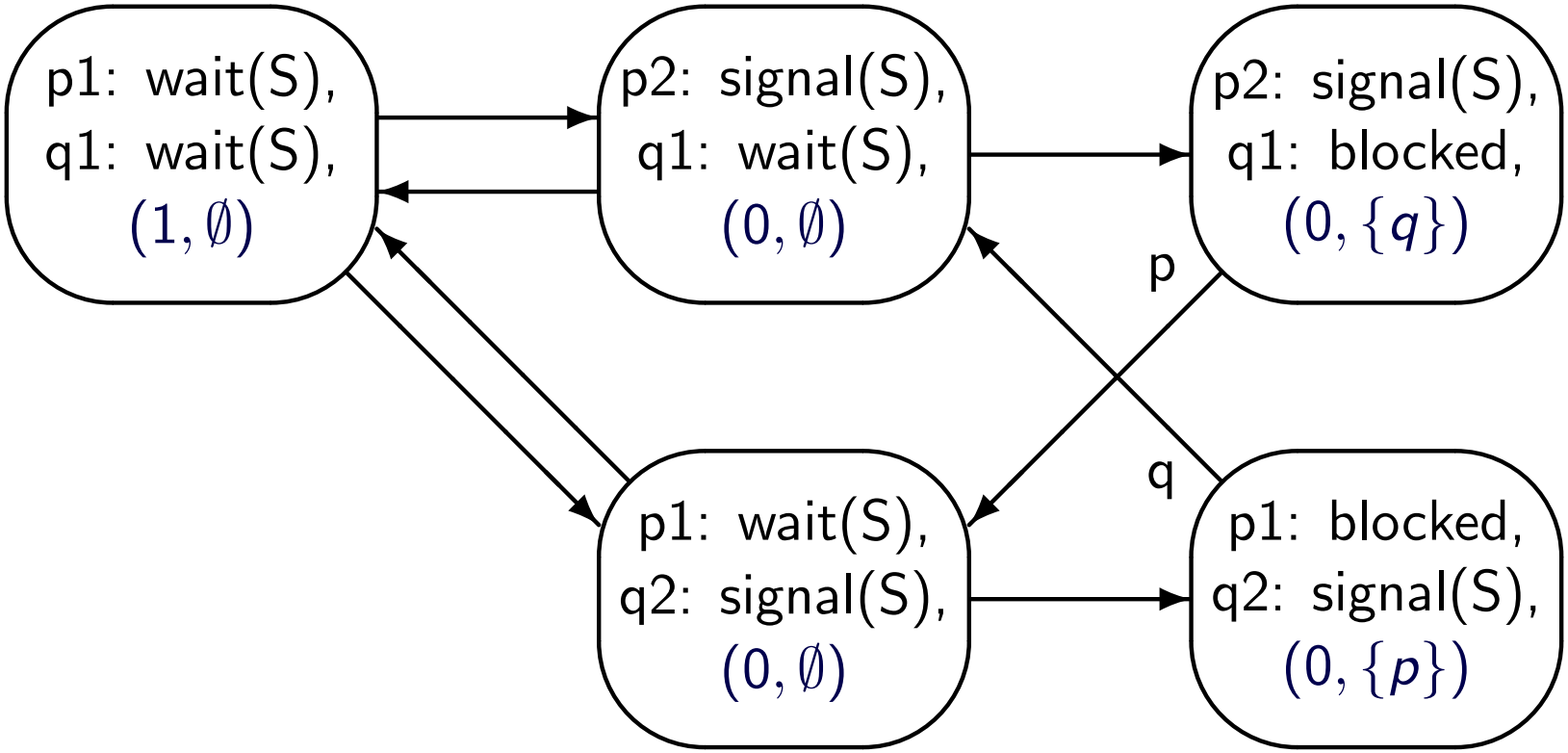
```
1   inline wait(S)   { d_step { S > 0 ; S-- } }
2   inline signal (S) { d_step{ S++ }}
```

This is known as a *busy-wait* semaphore.

Basic Uses of Semaphores (1): CSs

Algorithm 6.1: Critical section with semaphores (two processes)	
binary semaphore S $\leftarrow (1, \emptyset)$	
p	**q**
loop forever	loop forever
p1: non-critical section	q1: non-critical section
p2: wait(S)	q2: wait(S)
p3: critical section	q3: critical section
p4: signal(S)	q4: signal(S)

State Diagram for the Semaphore Solution



Algorithm 6.2: Critical section with semaphores (N proc.)

binary semaphore S $\leftarrow (1, \emptyset)$

```
    loop forever
p1:     non-critical section
p2:     wait(S)
p3:     critical section
p4:     signal(S)
```

Reasoning about semaphores

Proofs of safety properties of programs with semaphores often hinge on the appropriate exploitation of *semaphore invariants*. To express semaphore invariants, we let $\#\text{wait}(S)$ and $\#\text{signal}(S)$ denote the number of $\text{wait}(S)$, resp., $\text{signal}(S)$ operations executed so far. If S was initialised to $(k, \emptyset)$ then

$$v \geq 0 \tag{1}$$

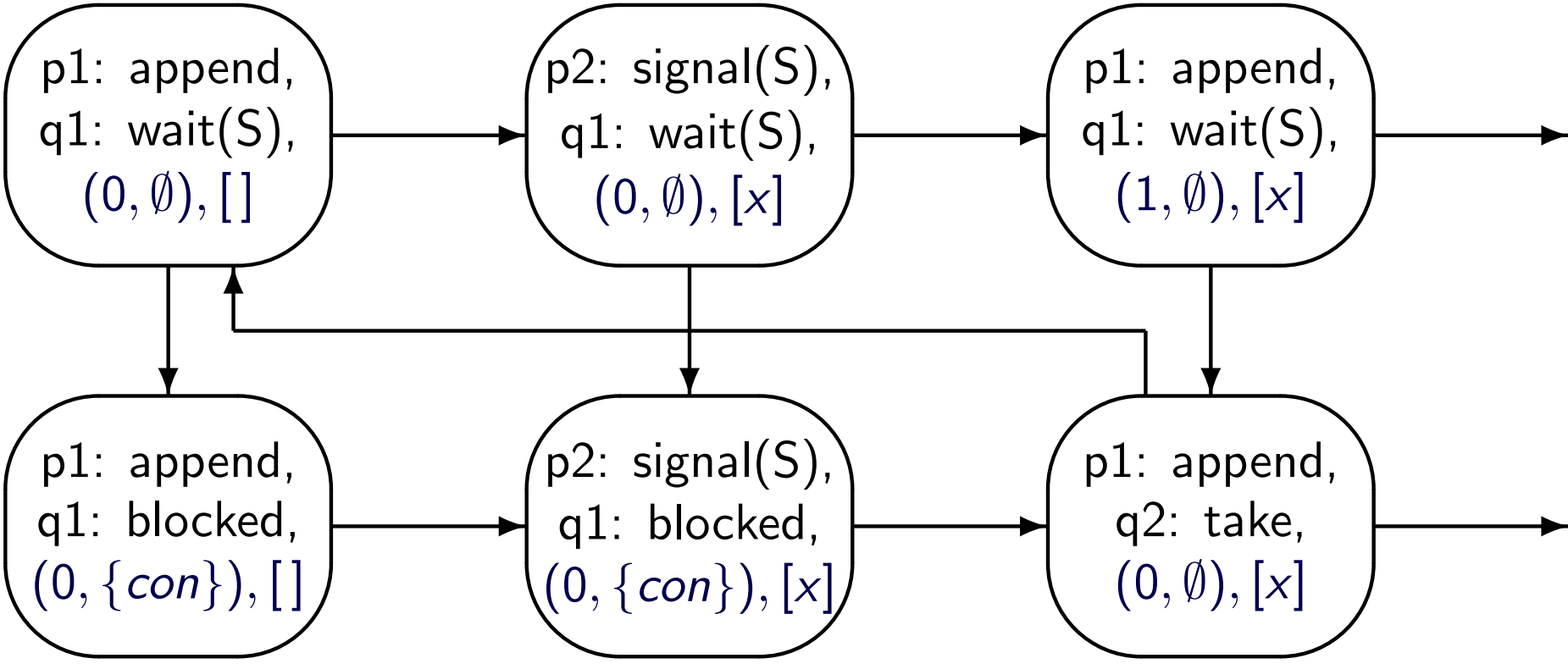
$$v = k + \#\text{signal}(S) - \#\text{wait}(S) \tag{2}$$

Algorithm 6.3: Producer-consumer (infinite buffer)

queue[T] buffer ← empty queue
semaphore full ← (0, ∅)

	producer		**consumer**
	T d		T d
	loop forever		loop forever
p1:	d ← produce	q1:	wait(full)
p2:	append(d, buffer)	q2:	d ← take(buffer)
p3:	signal(full)	q3:	consume(d)

Partial State Diagram for Producer-Consumer with Infinite Buffer



Algorithm 6.4: Producer-consumer (finite buffer, semaphores)

```
bounded[N] queue[T] buffer ← empty queue
semaphore full ← (0, ∅)
semaphore empty ← (N, ∅)
```

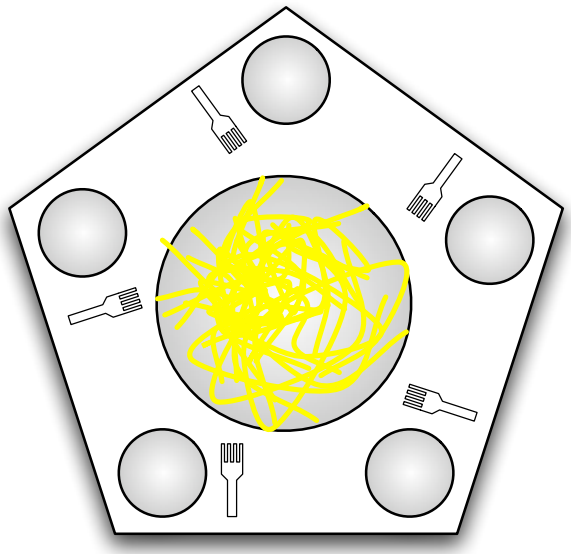
producer	**consumer**
T d	T d
loop forever	loop forever
p1: d ← produce	q1: wait(full)
p2: wait(empty)	q2: d ← take(buffer)
p3: append(d, buffer)	q3: signal(empty)
p4: signal(full)	q4: consume(d)

The Dining Philosophers Problem

Problem

Five philosophers sit around a dining table with a huge bowl of spaghetti in the centre, five plates, and five forks, all laid out evenly. For whatever reason, philosophers can eat spaghetti only with two forks.[a] *The philosophers would like to alternate between eating and thinking.*

[a] This is obviously a poor adaptation of an old problem from the East where requiring two chopsticks is more convincing.



Algorithm 6.5: Dining philosophers (outline)

```
      loop forever
p1:       think
p2:       preprotocol
p3:       eat
p4:       postprotocol
```

Same structure as the CS problem but with each fork being a critical section.

Algorithm 6.6: Dining philosophers (first attempt)

```
        semaphore array [0..4] fork ← [1,1,1,1,1]
        loop forever
p1:         think
p2:         wait(fork[i])
p3:         wait(fork[i+1])
p4:         eat
p5:         signal(fork[i])
p6:         signal(fork[i+1])
```

Algorithm 6.7: Dining philosophers (second attempt)

```
        semaphore array [0..4] fork ← [1,1,1,1,1]
        semaphore room ← 4
    loop forever
p1:     think
p2:     wait(room)
p3:     wait(fork[i])
p4:     wait(fork[i+1])
p5:     eat
p6:     signal(fork[i])
p7:     signal(fork[i+1])
p8:     signal(room)
```

Algorithm 6.8: Dining philosophers (third attempt)

```
        semaphore array [0..4] fork ← [1,1,1,1,1]
        philosopher 4
        loop forever
p1:         think
p2:         wait(fork[0])
p3:         wait(fork[4])
p4:         eat
p5:         signal(fork[0])
p6:         signal(fork[4])
```

Main Disadvantages of Semaphores

1. *Lack of structure*: when building a large system, responsibility is diffused among implementers.
   Someone forgets to call signal $\Longrightarrow$ possible deadlock.
2. *Global visibility*: when something goes wrong, the whole program must be inspected $\Longrightarrow$ deadlocks are hard to isolate.

Solution: *monitors* concentrate one responsibility into a single module and encapsulate critical resources.

Monitors

more structure than semaphores; more control than await

History:

- Hoare's 1974 paper
- languages — Concurrent Pascal (1975)... Java, Pthreads library

Generalise objects:

- data encapsulation—all fields are private
- implicit mutual exclusion—procedure invocations are *implicitly atomic*
- explicit signaling

Algorithm 6.9: Atomicity of monitor operations

```
monitor CS
    integer n ← 0

    operation increment
        integer temp
        temp ← n
        n ← temp + 1
```

p	q
p1: loop ten times	q1: loop ten times
p2: CS.increment	q2: CS.increment

Program structure

$$\text{monitor}_1 \ \ldots \text{monitor}_M$$

$$\text{process}_1 \ \ldots \text{process}_N$$

- processes interact indirectly by using the same monitor
- processes call monitor procedures
- at most one call active in a monitor at a time — by definition
- explicit signaling using condition variables
- *monitor invariant*: predicate about local state that is true when no call is active

Condition variables

are *named FIFO queues* of blocked processes:

Ben-Ari writes: condition cv

Processes executing a procedure of that monitor can

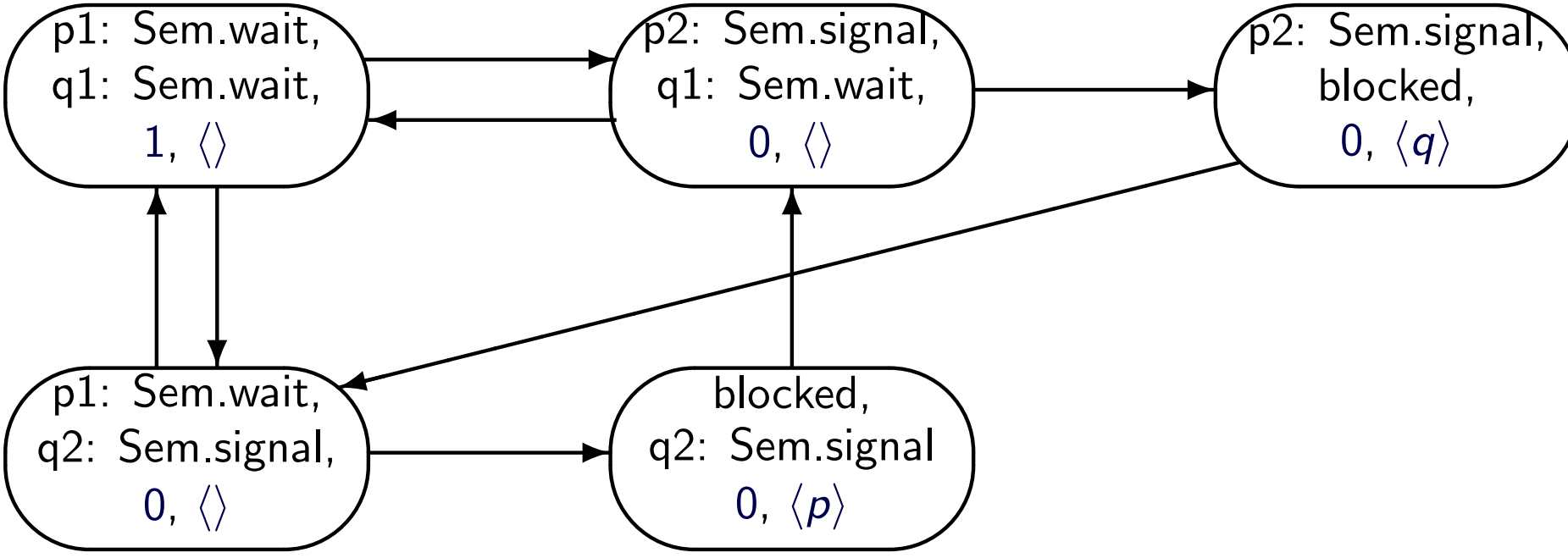
- voluntarily suspend themselves using waitC(cv),
- unblock the first suspended process by calling signalC(cv), or
- test for emptiness of the queue: empty(cv).

Algorithm 6.10: Semaphore simulated with a monitor

```
monitor Sem
    integer s ← k
    condition notZero
    operation wait
        if s = 0
            waitC(notZero)
        s ← s − 1
    operation signal
        s ← s + 1
        signalC(notZero)
```

p	q
loop forever	loop forever
non-critical section	non-critical section
p1: Sem.wait	q1: Sem.wait
critical section	critical section
p2: Sem.signal	q2: Sem.signal

State Diagram for the Semaphore Simulation



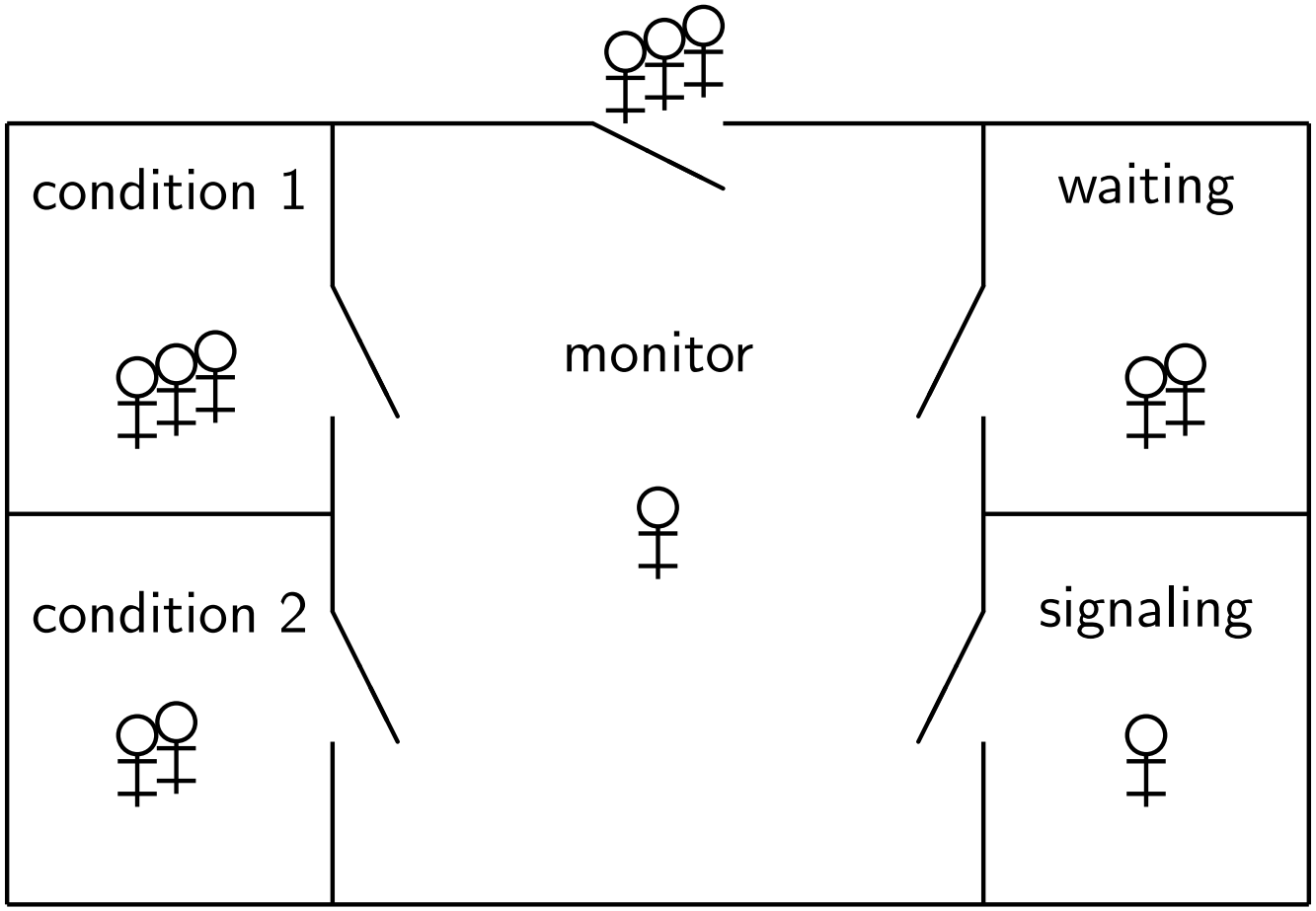
Algorithm 6.11: Producer-consumer (finite buffer, monitor)

```
monitor PC
    bufferType buffer ← empty
    condition notEmpty
    condition notFull
    operation append(datatype V)
        if buffer is full
            waitC(notFull)
        append(V, buffer)
        signalC(notEmpty)
    operation take()
        datatype W
        if buffer is empty
            waitC(notEmpty)
        W ← head(buffer)
        signalC(notFull)
        return W
```

Algorithm 6.11: Producer-consumer . . . (continued)

producer		consumer	
	datatype D		datatype D
	loop forever		loop forever
p1:	D ← produce	q1:	D ← PC.take
p2:	PC.append(D)	q2:	consume(D)

The Immediate Resumption Requirement



Signaling disciplines

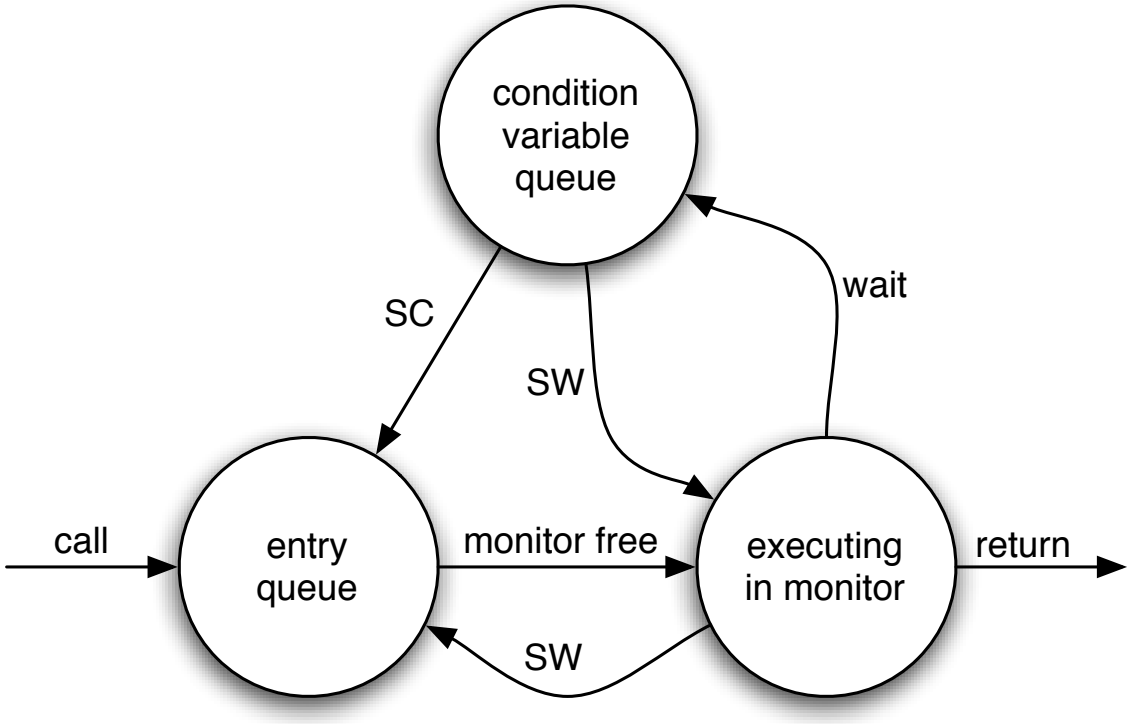
Precedences:

- S the signaling process
- W waiting on a condition variable
- E waiting on entry

signal and urgent wait: $E < S < W$ (Hoare's paper)

signal and continue $E = W < S$ (Java, pthreads)

Diagram for monitors



Simulating Monitors in Promela 1

```
1   bool lock = false;
2
3   typedef Condition {
4       bool gate;
5       byte waiting;
6   }
7   inline enterMon() {
8       atomic {
9           !lock;
10          lock = true;
11      }
12  }
13  inline leaveMon() {
14      lock = false;
15  }
```

Simulating Monitors in Promela 2

```
1   inline waitC(C) {
2       atomic {
3           C.waiting++;
4           lock = false;       /* Exit monitor */
5           C.gate;             /* Wait for gate */
6           lock = true;        /* IRR */
7           C.gate = false;     /* Reset gate */
8           C.waiting--;
9       }
10  }
```

Simulating Monitors in Promela 3

```
1   inline signalC(C) {
2       atomic {
3           if
4               /* Signal only if waiting */
5           :: (C.waiting > 0) ->
6              C.gate = true;
7              !lock;          /* IRR - wait for released lock */
8              lock = true; /* Take lock again */
9           :: else
10          fi ;
11      }
12  }
13
14  #define emptyC(C) (C.waiting == 0)
```

Algorithm 6.12: Readers and writers with a monitor

```
monitor RW
    integer readers ← 0
    integer writers ← 0
    condition OKtoRead, OKtoWrite
    operation StartRead
        if writers ≠ 0 or not empty(OKtoWrite)
            waitC(OKtoRead)
        readers ← readers + 1
        signalC(OKtoRead)
    operation EndRead
        readers ← readers − 1
        if readers = 0
            signalC(OKtoWrite)
```

Algorithm 6.12: Readers and writers with a monitor (continued)

```
operation StartWrite
    if writers ≠ 0 or readers ≠ 0
        waitC(OKtoWrite)
    writers ← writers + 1

operation EndWrite
    writers ← writers − 1
    if empty(OKtoRead)
        then signalC(OKtoWrite)
        else signalC(OKtoRead)
```

reader	writer
p1: RW.StartRead	q1: RW.StartWrite
p2: read the database	q2: write to the database
p3: RW.EndRead	q3: RW.EndWrite

POSIX threads

What? a widely available, standardized C library for multi-threaded programming

How?

1. include standard headers:

   #include <pthread.h>

2. create some threads by (repeatedly) calling

   ```
   int pthread_create (pthread_t *thread,
                       pthread_attr_t *attr,
                       void *(*start_routine )(void *),
                       void *arg);
   ```

   If attr is NULL, default attributes are used, eg on linux, the thread is *joinable* and has non-real-time scheduling policy.

Thread Attributes

Upon thread creation one needs to decide whether the creating thread wants to be able to *synchronize* with the termination of the created thread (which is the default behaviour) using

```
int pthread_join (pthread_t th, void **t_ret );
```

which "suspends the execution of the calling thread until the thread identified by th terminates, either by calling pthread_exit or by being cancelled."

Pitfall

When a joinable thread terminates, its memory resources (thread descriptor and stack) are not deallocated until another thread performs pthread_join on it. Therefore, pthread_join must be called once for each joinable thread created to avoid memory leaks.

If synchronization is not required, save a few resources and avoid the pitfall:

```
pthread_attr_t    attr ;
pthread_attr_init     (&attr);
pthread_attr_setdetachstate
(&attr,  PTHREAD_CREATE_DETACHED);
```

Examples

- simple.c: shows the basic structure
- pc.busy.c: producer/consumer using busy waiting and a single buffer
  pcn.busy.c: producer/consumer using busy waiting and an n-place buffer

Semaphores in POSIX threads

```
#include <semaphore.h>

int  sem_init (sem_t *sem, int  pshared,
               unsigned int value);
int  sem_wait(sem_t *sem);    /* P(sem) */
int  sem_trywait(sem_t *sem);
int  sem_post(sem_t *sem);    /* V(sem) */
int  sem_getvalue(sem_t *sem, int  *sval );
int  sem_destroy(sem_t *sem);
```

If pshared is zero then the sem is local to the current process, otherwise it is to be shared between several processes.

Examples

pc.sems.c: producer/consumer using semaphores and a single buffer

pcn.sems.c: producer/consumer using semaphores and an n-place buffer

Locks & Monitors in POSIX Threads

In pthreads, a basic lock is called a *mutex*, for **mut**ual **ex**clusion device.

Such a device is in one of two states:

unlocked $\simeq$ not owned by any thread

locked $\simeq$ owned by one thread

A thread attempting to *lock* a mutex already owned by another thread is *suspended*.

A monitor procedure is simulated using pthreads by bracketing the procedure code with mutex lock/unlock.

Details: RTFM

Static initialization:

```
pthread_mutex_t fastmutex =
  PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recmutex =
  PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
pthread_mutex_t errchkmutex =
  PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

```
int  pthread_mutex_init (     pthread_mutex_t      *mutex,
                        const pthread_mutexattr_t *mutexattr);
int  pthread_mutex_lock(     pthread_mutex_t      *mutex);
int  pthread_mutex_trylock(pthread_mutex_t        *mutex);
int  pthread_mutex_unlock( pthread_mutex_t        *mutex);
int  pthread_mutex_destroy(pthread_mutex_t        *mutex);
```

pthread_mutex_lock behaviour depends on mutex kind. RTFM.

Condition variables

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init (pthread_cond_t *cond,
                       pthread_condattr_t *cond_attr);
int pthread_cond_signal (pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex,
                      const struct timespec *abstime);
int pthread_cond_destroy (pthread_cond_t *cond);
```

ZZZZZzzzzzzzzzzzzzzzzzzzz

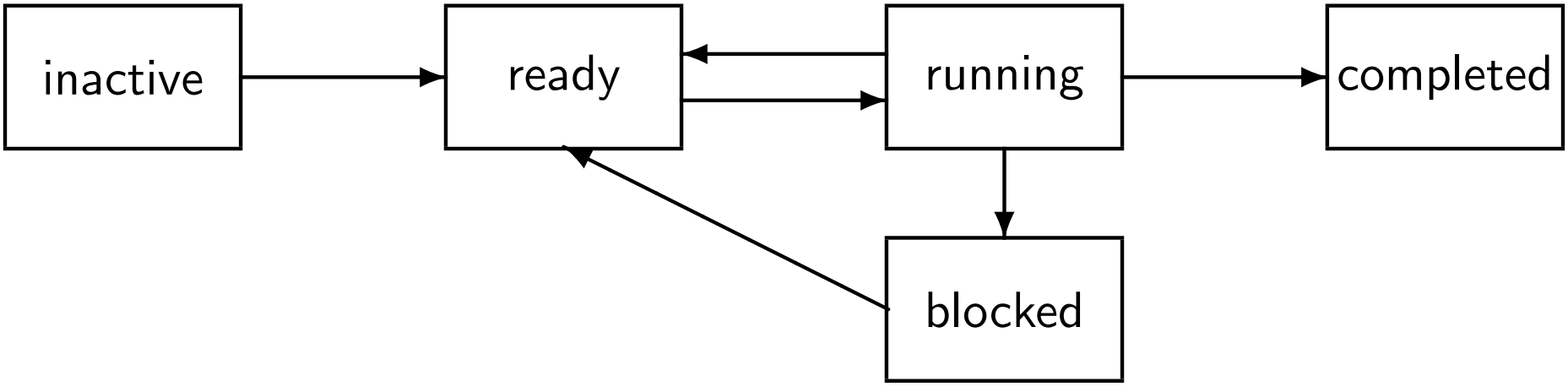
Example

summing the elements of a matrix uses a thread for each strip

See matrix.sum.c

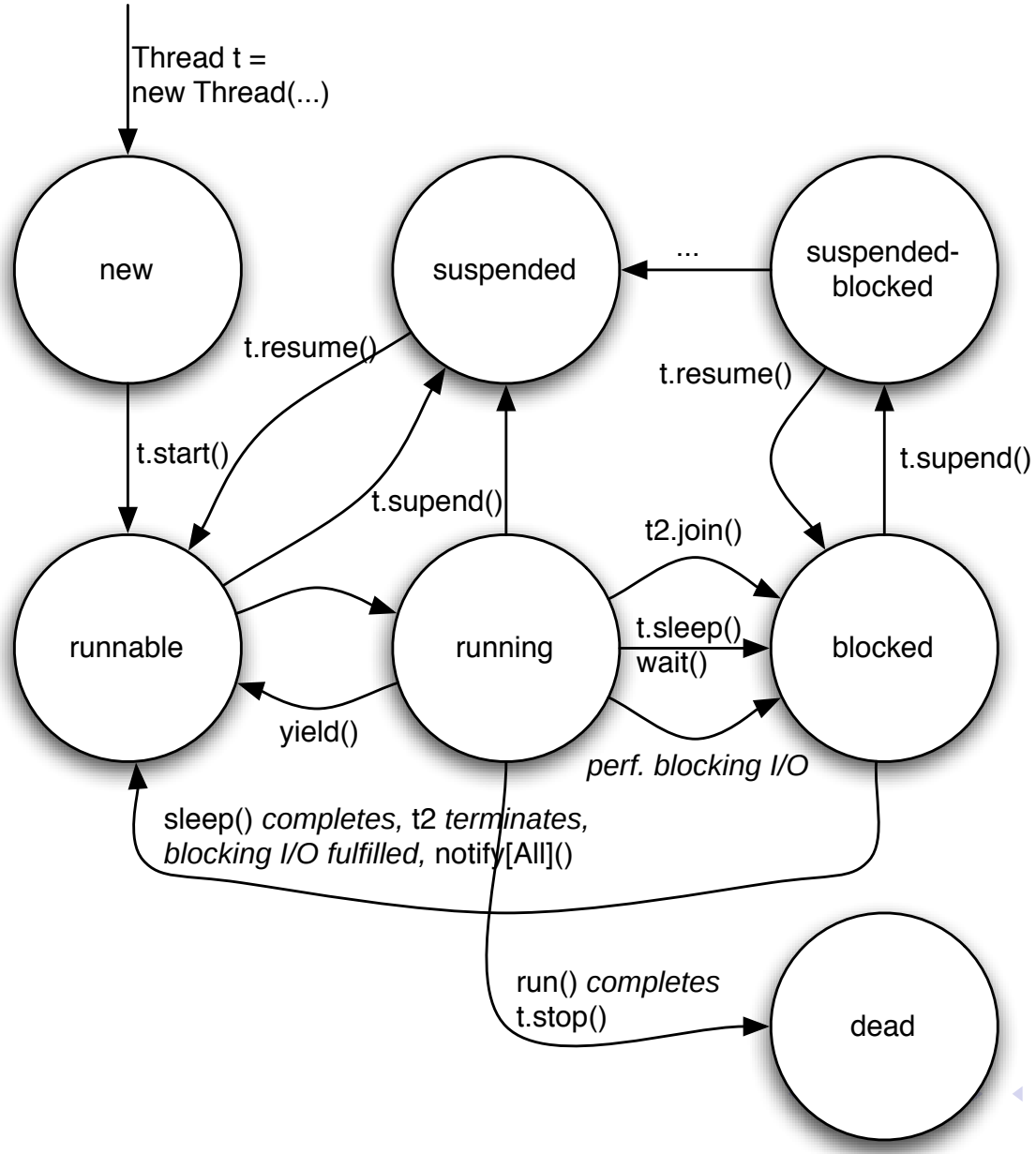
Process states

For our purposes it suffices to consider a process as being in one of five states:



while in some version of reality. . .

Java process states



Thread state

A Java thread object t always is in one of 7 possible states:

new – just created Thread t = new Thread(...);

runnable – entered when t.start () is called. All runnable threads are maintained in the *runnable set* or *ready queue* and take turns executing their run method.

running – actually executing; can get back to **runnable** by calling yield

suspended – entered when t.suspend(); is called; can get back by another thread calling t.resume()

But do read Oracle's gospel on deprecated thread primitives.

blocked – entered when

- `t.sleep()` is called,
- it calls `wait()` inside a `synchronized` method,
- it calls `join()` on some thread object that hasn't, terminated yet, or
- it performs blocking I/O

and exited to enter the **runnable** state when

- its `sleep` method call completes,
- it joins with a terminated thread,
- its blocking I/O request is fulfilled by the OS, or
- some other thread calls `notify` or `notifyAll` in a `synchronized` method of the object `t` earlier called `wait`.

suspended-blocked – suspended while blocked...

dead – entered when `run` completes or `t.stop()` is called.

Execution model

Each thread t has a *priority*, either inherited from its creator thread or set explicitly using t.setPriority (..) . The Java thread scheduler ensures that one of the highest priority threads executes. It even *preempts* a lower priority **running** thread when a higher priority thread becomes **runnable**.

All highest priority runnable threads share the process's CPU time in a round-robin fashion executing for time slices in the order of 100ms.