TDDB68 + TDDE47 + TDDD82

Lecture:
Synchronisation

Mikael Asplund
Real-time Systems Laboratory
Department of Computer and Information Science

Copyright Notice:
Thanks to Christoph Kessler and Simin Nadjm-Tehrani for much of the material behind these slides.

The lecture notes are partly based on Silberschatz's, Galvin's and Gagne's book ("Operating System Concepts", 7th ed., Wiley, 2005). No part of the lecture notes may be reproduced in any form, due to the copyrights reserved by Wiley. These lecture notes should only be used for internal teaching purposes at the Linköping University.

Reading guidelines

- Silberschatz, Galvin and Gagne, Operating System Concepts
  - 9th edition: Chapter 6.1-6.9
  - 10th edition: Chapter 6.1-6.7 + 7.1-7.3

Recap

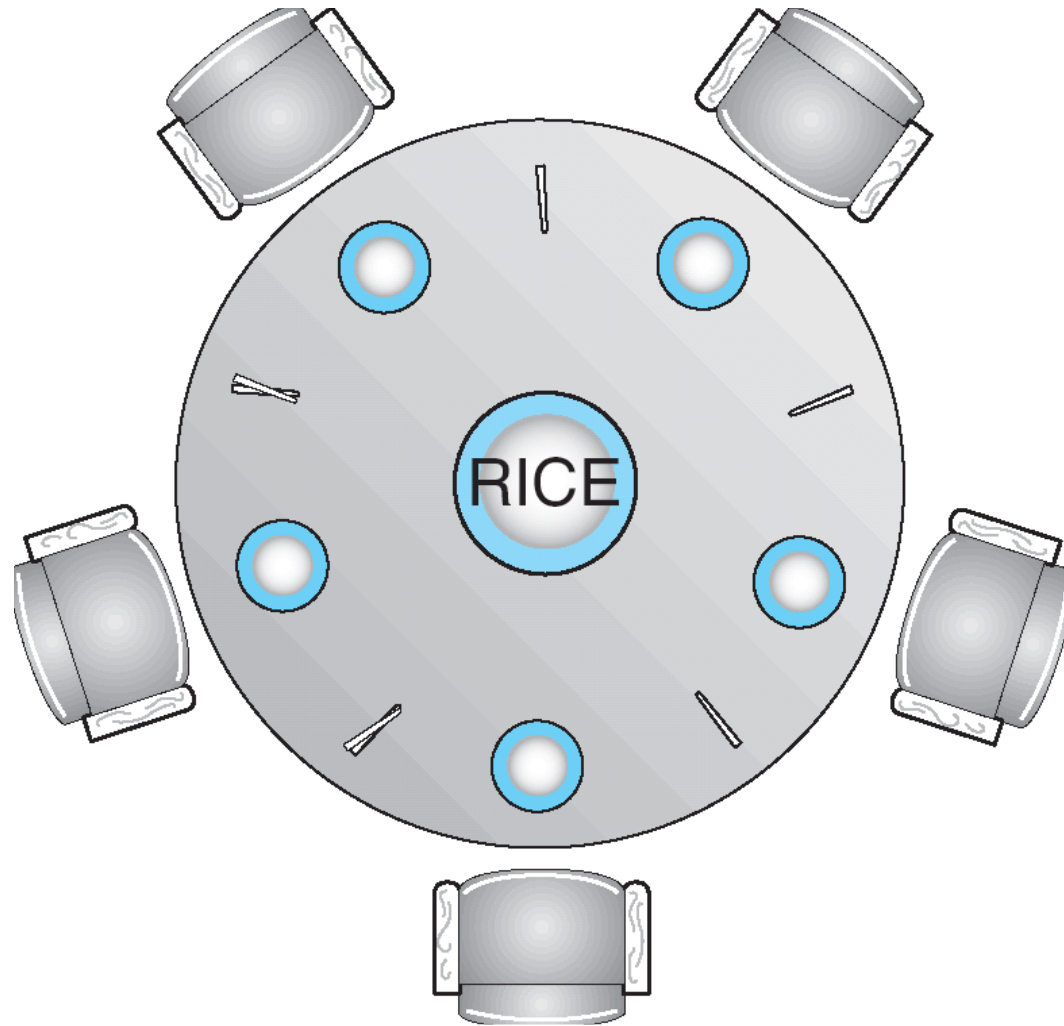
- Concurrent processes can communicate through message passing or shared memory
  - **Message passing:** clean but sometimes complex and slow
  - **Shared memory:** fast but requires protecting data

- Concurrent programs must avoid race conditions

General problems

- **Conditional action**
  - Examples:
    - Compute the interest when all transactions have been processed
    - Book a flight seat only if seats are available
- **Mutual exclusion**
  - Example:
    - Two customers shall not be booked on the same seat

Dining-Philosophers Problem



Critical section

- Consider n processes that need to exclude concurrent execution of some parts of their code

```
Process Pi {
    entry-protocol
    critical-section
    exit-protocol
    non-critical-section
  }
```

- Fundamental problem to design entry and exit protocols for critical sections

Critical-Section Problem

- Mutual Exclusion
- Progress
- Bounded waiting

Mutual Exclusion

If process P is executing in critical section C, then no other processes can be executing in C (accessing the same shared data/resource).

Progress

If no process is executing in critical section C and there exist some processes that wish to enter C, then the selection of the process that will enter C next cannot be postponed indefinitely.

Bounded waiting

A bound must exist on the number of times that other processes are allowed to enter critical section C after a process has made a request to enter C and before that request is granted.

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the N processes

First attempt

```
Process P1
while (true) {
  flag1 = up
  while (flag2 == up) {
    // do nothing
  }
  critical section
  flag1 = down
  non-critical section
}
```

```
Process P2
while (true) {
  flag2 = up
  while (flag1 == up) {
    // do nothing
  }
  critical section
  flag2 = down
  non-critical section
}
```

[Dijkstra 1965]

Menti.com: 19 00 59

Second attempt

```
Process P1
while (true) {
   while (flag2 == up) {
     //do nothing
  }
  flag1 = up
  critical section
  flag1 = down
  non-critical-section
}
```

```
Process P2
while (true) {
   while (flag1 == up) {
     //do nothing
  }
  flag2 = up
  critical section
  flag2 = down
  non-critical-section
}
```

Menti.com: 19 00 59

Third attempt

Process P1

```
while (true) {
  while (turn == 2) {
    //do nothing (busy
   waiting)
  }
  critical section
  turn = 2
  non-critical-section
}
```

Process P2

```
while (true) {
  while (turn == 1) {
    //do nothing (busy
   waiting)
  }
  critical section
  turn = 1
  non-critical-section
}
```

Menti.com: 91 61 64

Peterson's algorithm

```
Process P1
while (true) {
   flag1 = up
   turn = 2
   while (flag2 == up) and
         (turn == 2) {
     //do nothing
   }
   critical section
   flag1 = down
   non-critical-section
}
```

```
Process P2
while (true) {
   flag2 = up
   turn = 1
   while (flag1 == up) and
         (turn == 1) {
     //do nothing
   }
   critical section
   flag2 = down
   non-critical-section
}
```

Menti.com: 91 61 64

Summary of early solutions

- Implementing synchronisation and concurrency with shared variables, using only sequential programming constructs, is difficult and error prone

- But...
  - They do not require any support

Hardware Support for Synchronization

- **TestAndSet**: test memory word and set value atomically
- **Swap**: swap contents of two memory words atomically

If multiple TestAndSet or Swap instructions are executed simultaneously (each on a different CPU in a multiprocessor), then they take effect sequentially in some arbitrary order.

TestAndSet Instruction

- Definition in pseudocode:

```
boolean TestAndSet (boolean *target) {
    boolean rv = *target
    *target = true
    return rv        // return the OLD value
 }
```

Solution using TestAndSet

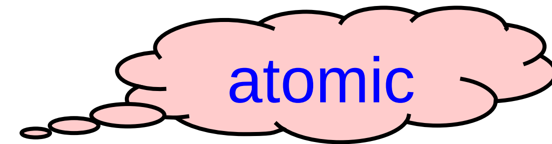
```
lock = false //shared variable

while (true) {
  while (TestAndSet (&lock)) {
      // do nothing (busy waiting)
  }
  critical section
  lock = false
  non-critical section
}
```

Swap Instruction

- Definition in pseudocode:

```
void Swap (boolean *a, boolean *b) {
        boolean temp = *a
        *a = *b
        *b = temp
    }
```



Solution using Swap

```
lock = false          //shared variable

while (true) {
  tmp = true;      //local variable (not shared)
  while ( tmp == true) {
    Swap (&lock, &tmp );       // busy waiting…
  }
  critical section
  lock = false;
  non-critical section
}
```

Programming language support

- Would be useful to have support from an operating system or a programming language

- Modern programming languages have explicit support:
  - java.util.concurrency provides good support.
  - Ada: built-in run-time support with explicit task synchronisation entry points (Rendezvous)
  - Python: threading import *
  - ...

Next: more support

How can the mutual exclusion and synchronisation problems be solved with other constructs?

- Semaphore, lock and monitor

- What do we gain by using the support in OS or programming language?

Semaphores

- A semaphore S is a *non-negative* integer variable on which only two atomic operations wait and signal can be performed

```
wait(S):
   wait until S > 0
   S = S-1

signal(S):
    S = S+1
```

Solution with semaphore

```
semaphore S = 1

while (true) {
  wait(S)
  critical section
  signal(S)
  non-critical section
}
```

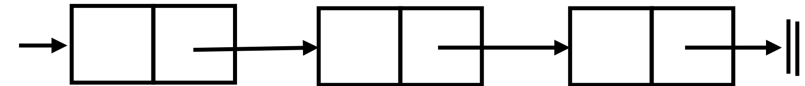
Atomicity of semaphore implementation must be provided by the supporting environment

Spin locks

- All entry protocols so far (including semaphore wait) uses a busy wait loop – called a spin lock
- Sometimes necessary (kernel-level programming)
- Wasteful for synchronization of user processes

Eliminate Busy Waiting

- With each semaphore there is an associated **waiting queue**. Each entry in a waiting queue contains:
  - Process table index, e.g. pid
  - Pointer to next entry



- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue.
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue.

Samaphore datastructure with a queue

```
typedef struct {
    int value;
    struct process *wqueue;
} semaphore;
```

Wait implementation w/o busy waiting

```
void wait ( semaphore *S ) {
   S->value--;
    if (S->value < 0) {
        add this process to S->wqueue;
        block();  // I release the lock on the critical
     }            // section for S and release the CPU
  }
```

Signal Implementation w/o busy waiting

```
void signal ( semaphore *S ) {
  S->value++;
  if (S->value <= 0) {
    remove a process P from S->wqueue;
   wakeup (P);  // append P to ready queue
  }
}
```

Counting semaphores

- When more than one instance of a resource is available, e.g. print servers
- Processes can use up to max available but no more
- The semaphore can be initialised to provide access for n processes

Semaphore initialization

- Crutial to determine the semantics of the semaphore

- Must be stated in your exam answers!

A semaphore with maximum value 1 is called a **binary semaphore**, useful to implement lock.

Locks

- Binary semaphore

- Operations often called
  - **Acquire** (instead of wait)
  - **Release** (instead of signal)

- Only the thread that acquired the lock can release it!

Problems

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes
- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Sources of synchronization errors

- Omitting **wait** (mutex) or **signal** (mutex) (or both)
- **wait** (mutex) …. **wait** (mutex)
- **wait** (mutex1) …. **signal** (mutex2)
- Multiple semaphores with different orders of **wait**() calls
  - Example: Each philosopher first grabs the chopstick to its left → risk for deadlock!

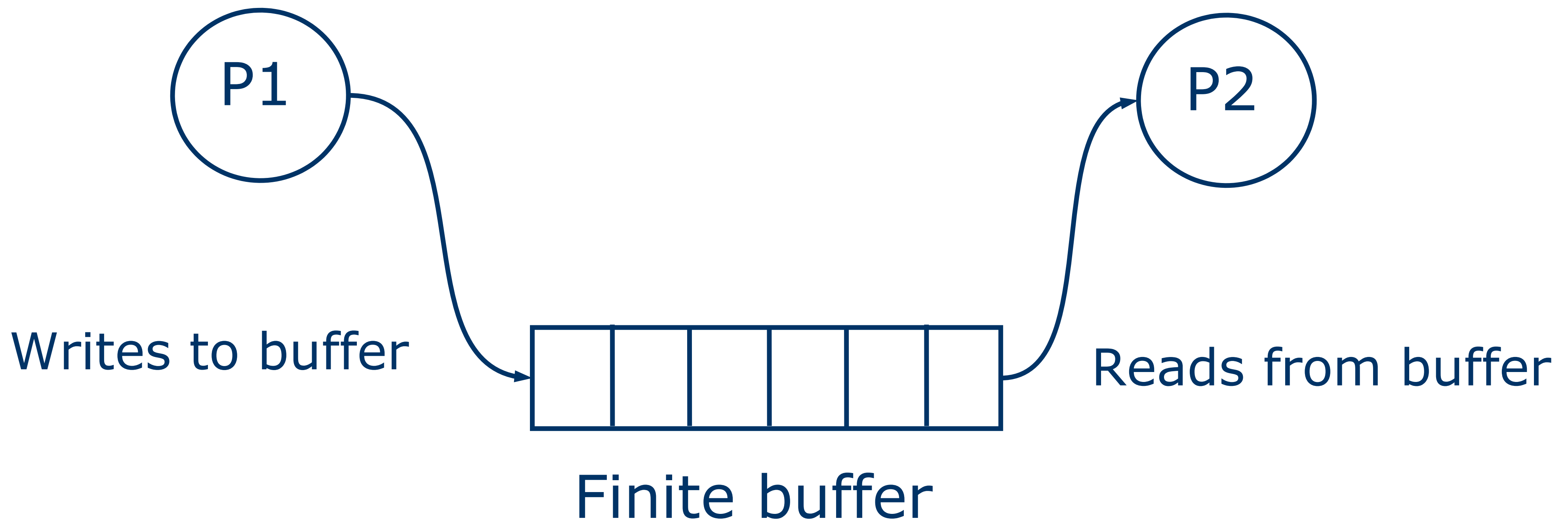
Focus on the resource (data)!

- Non-shared data does not need protecting
  - Automatic (stack, local) variables
- Same resource must be protected with the same synchronization primitive
- Consistency requirements and access patterns determine the granularity of synchronization

Complex data structures

Decoupling process rates

- Finite buffers



Issues

- Writing to full buffer
- Reading from empty buffer
- Two write operations to the same element

Condition variables

- Declared as special synchronisation variables:

  condition X;

- With two designated operations:

  wait: suspend the calling process

  signal/notify: if there are suspended processes on this variable, wake one up

Example!

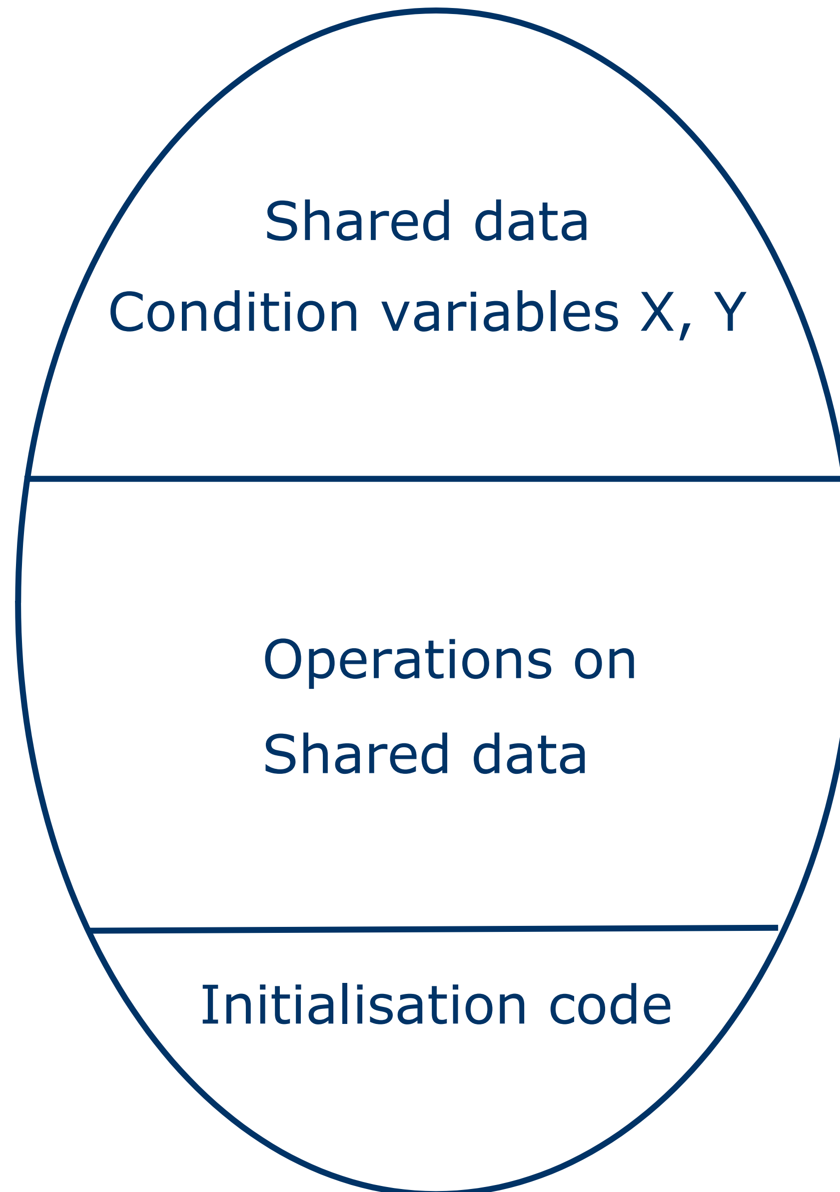
What is a monitor?

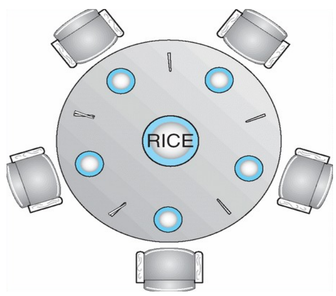
- A programming abstraction consisting of:
  - A data structure on which programmer can define operations – which can only be run one at a time
  - Condition variables for synchronisation

- Encapsulates shared data that several processes can operate upon
- All access is with mutual exclusion
- Pre object-orientation!

[Hoare 74]

Monitor overview





Monitor Solution to Dining Philosophers

```
monitor DP {
  enum {THINKING, HUNGRY, EATING} state
   [5];
  condition self [5];

  void pickup ( int i ) {
    state[i] = HUNGRY;
    test ( i );
    if (state[i] != EATING)
      self [i].wait();
  }

  void putdown ( int i ) {
    state[i] = THINKING;
    test((i+4)%5);  // left neighbor
    test((i+1)%5);  // right neighbor
  }

...
```

```
...
  void test ( int i ) {
    if ((state[(i+4)%5] != EATING)
          && (state[i] == HUNGRY)
          && (state[(i+1)%5] !=
   EATING)) {
      state[i] = EATING ;
      self[i].signal () ;
    }
  }

  initialization_code() {
    for (int i = 0; i < 5; i++) {
      state[i] = THINKING;
    }
  }
}
```

Observations

- Programmer uses wait and signal inside the code that applies the operations on the shared data structure

Note:

- The condition variable has no values assigned to it
- The queue associated with each variable is the main synchronisation mechanism
- Different semantics from semaphore operations for wait and signal

Process queues

Queue of processes wanting to execute some monitor operation

Shared data

Condition variables X, Y

X:

Y:

queue of processes waiting for X

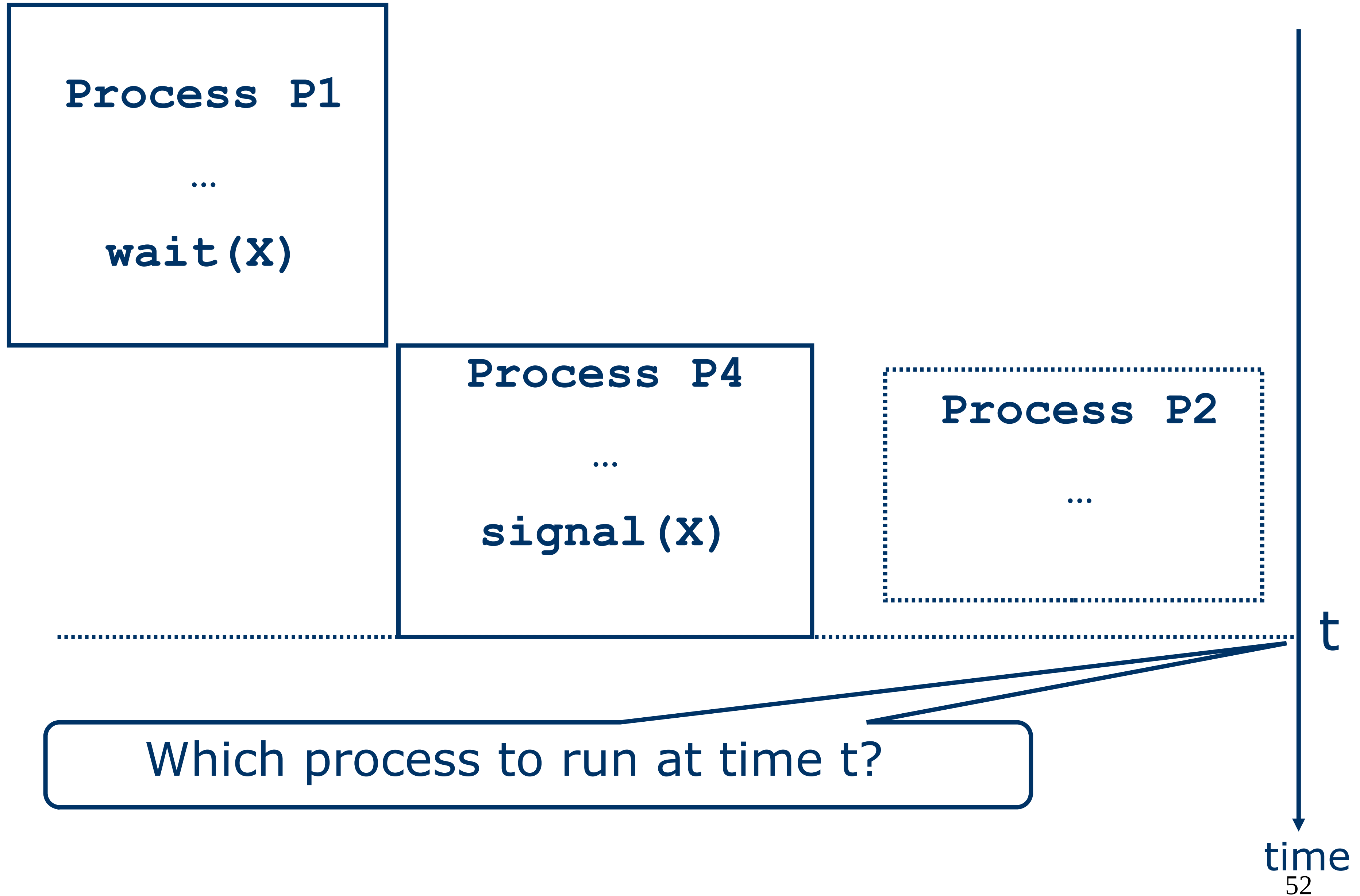
queue of processes waiting for Y

Operations on Shared data

Initialisation code

These operations may use wait/signal on X, Y

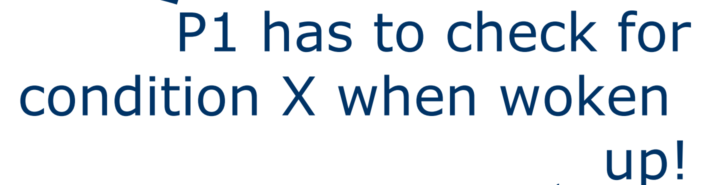
How does it work?



Options

- Original Hoare monitor: let the woken up process (P1) continue

- Pragmatic solution (Java): let the signalling process continue, and wake up P1 once P4 is suspended/exits



What happens if a process that holds a semaphore/monitor is killed?

Lock-free algorithms!

Update: Some available tools

- ThreadSanitizer
  - Part of clang and gcc
  - Finds unprotected data in C/C++
  - Requires instrumented code
- Intel Inspector
  - Powerful but proprietary
- Visual Studio
  - Concurrency check rules