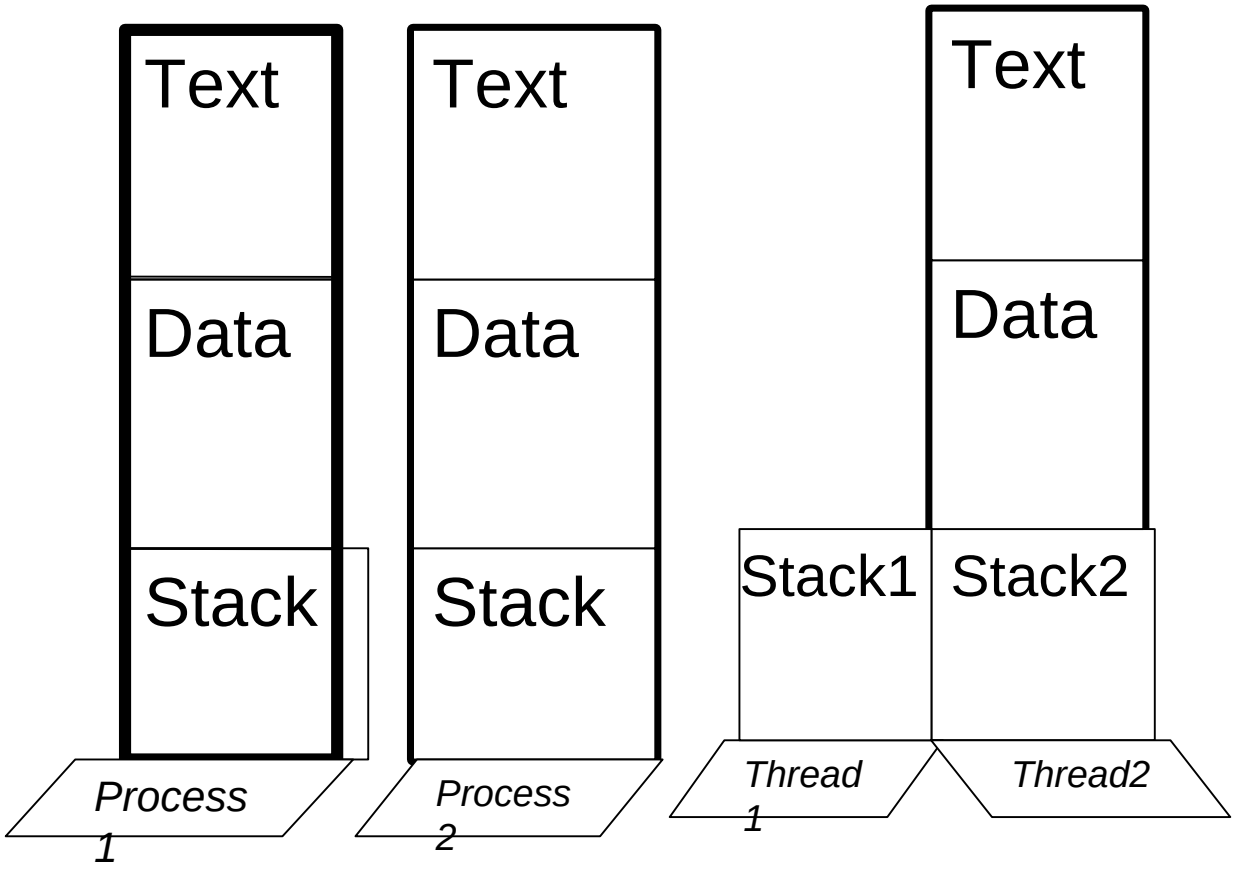
Threads

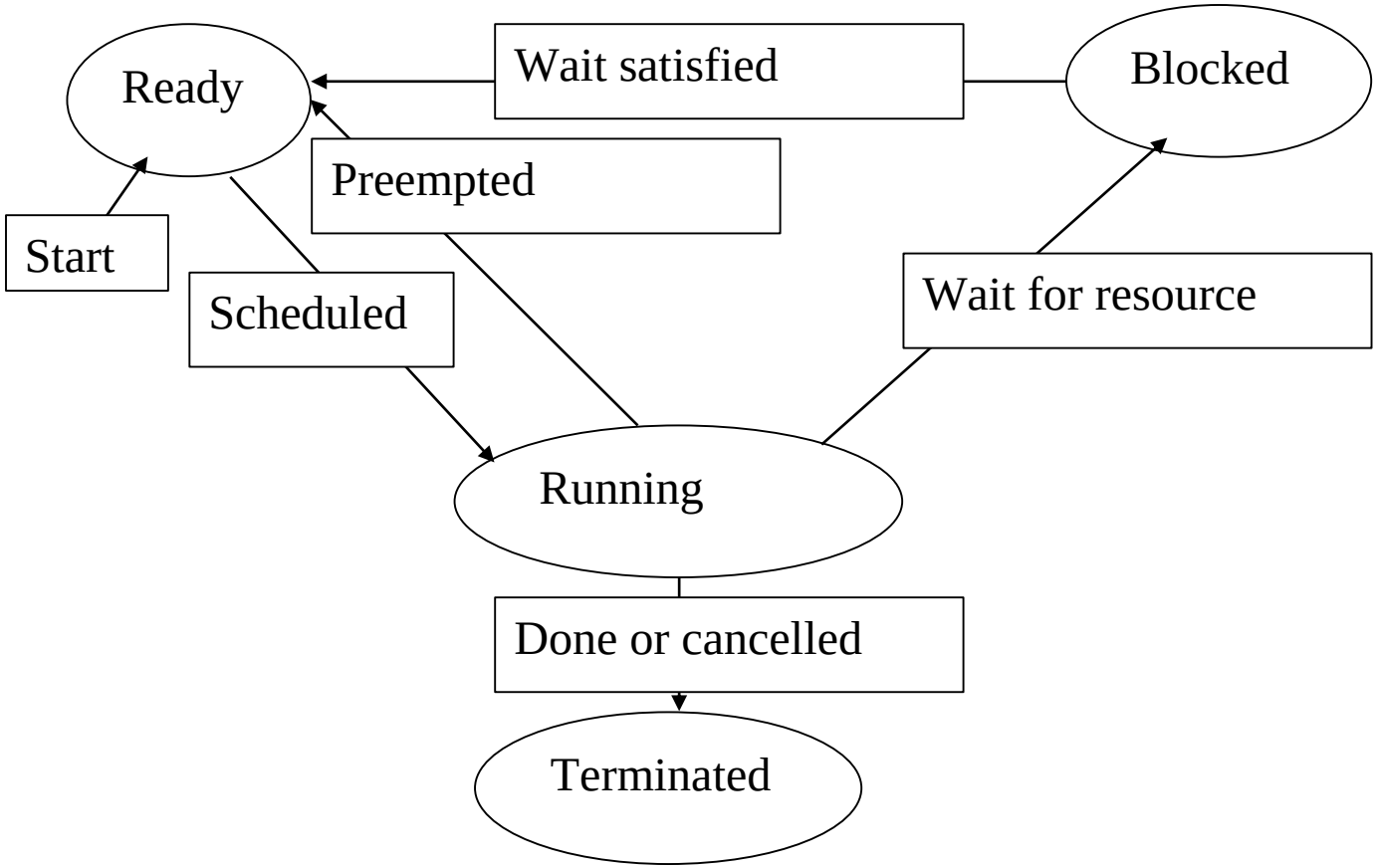
Threads

Process: what operating system does with a program

Thread: flow of control in a process (with own stack)



State diagram of a thread is similar to state diagram of a process



Threads

What threads are for:

• expressing something what cannot be expressed otherwise, i.e. a reaction which is independent of computation in a process (GUI, for example)

What threads have **not** been meant for:

• anything else

Threads

Contemporary trends of using threads:

- implementation of GUI
- kicking more (?) performance out of PCs (web servers, numeric computation, games, ...)
- production of specialised hardware (e.g. Nvidia/CUDA)

“Multicore processors are everywhere“, „Gain performance by harnessing threads in servers“, ...

Critical sections

Whenever two threads access the same shared variable at the same time, there is a problem.

For example, thread T1 increments an integer s, T2 decrements it (let s be 0 before that):

T1: s = s + 1;

T2: s = s - 1;

The problem is that the operations ++ and -- can be translated to the machine code as follows:

T1: LOAD reg1, s; INC reg1; STORE s, reg1;

T2: LOAD reg2, s; DEC reg2; STORE s, reg2;

The scheduler can execute these e.g. in the order

LOAD reg1, s; LOAD reg2, s; INC reg1; STORE s, reg1; DEC reg2; STORE s, reg2;

Both threads have been executed, but now s equals -1, not 0 as expected!

Critical sections, mutual exclusion

An access to a shared memory (any access, no matter whether it is a read or write) is called **a critical region** of a thread. The programmer must ensure that no two threads are in their critical regions at the same time

We will first show two classical methods which ensure **mutual exclusion**: Dekker's algorithm and Peterson's algorithm (to keep things simple, we present solutions for 2 threads). Both algorithms consists of two pieces of code (protocols): one on the entrance, the other one at the exit of the critical section. Every thread executes these protocols whenever it enters or leaves a critical section

Dekker's algorithm, wrong version

```
int in1 = in 2 = FALSE;
```

Thread 1:

```
/* entry */
in1 = TRUE;
while (in2)
{
  in1 = FALSE;
  /* delay */
  in1 = TRUE;
}
/* critical section */
...
/* exit */
in1 = FALSE;
```

Thread 2:

```
/* entry */
in2 = TRUE;
while (in1)
{
  in2 = FALSE;
  /* delay */
  in2 = TRUE;
}
/* critical section */
...
/* exit */
in2 = FALSE;
```

Why is this wrong?

Dekker's algorithm, correct version

```
int in1 = in2 = FALSE;
int turn = 1;
```

Thread 1:

```
in1 = TRUE;
while (in2)
{
  if (turn == 2)
  {
    in1 = FALSE;
    while (turn == 2)
      ;
    in1 = TRUE;
  }
}
/* critical section */
...
turn = 2;
in1 = FALSE;
```

Thread 2:

```
in2 = TRUE;  /* entry */
while (in1)
{
  if (turn == 1)
  {
    in2 = FALSE;
    while (turn == 1)
      ;
    in2 = TRUE;
  }
}
/* critical section */
...
turn = 1;        /* exit */
in2 = FALSE;
```

Peterson's algorithm

```
int in1 = FALSE; int in2 = FALSE; int last = 1;

Thread 1:
in1 = TRUE; last = 1; /* entry protocol for thread1 */
while (in2 && last == 1)
;
/* critical section */
...
in1 = FALSE; /* exit protocol for thread1 */

Thread 2:
in2 = TRUE; last = 2; /* entry protocol for thread1 */
while (in1 && last == 2)
;
/* critical section */
...
in2 = FALSE; /* exit protocol for thread1 */
```

Dekker's and Peterson's algorithms: ticket, bakery

Peterson’s and Decker's algorithm can be generalised for an arbitrary number of threads. Idea: each thread must pass (N-1) through phases of an entry protocol, where N is the number of threads. It is guaranteed that exactly 1 thread passes through N-1 phases in one moment. Each phase solves an instance of mutual exclusion problem for 2 threads

Alternative solutions (easier to understand): ticket algorithm, bakery algorithm

Sketch of the ticket algorithm: at entrance, thread „obtains a ticket” with a number. The number is automatically incremented. The entrance is clear for the thread holding the number “shown on a display“

Sketch of the bakery algorithm: at entrance, thread “looks” at the numbers of all waiting threads and chooses some greater number for itself. When this number becomes the least, entrance is clear

Dekker's and Peterson's algorithms: ticket, bakery

Problem of ticket algorithm: integers are finite, they will overflow

The same is true for the bakery algorithmus, but only when a thread is waiting at the entrance to the critical region

An additional requirement to the mechanism of mutual exclusion: **fairness**. It guarantees that every thread enters a critical region (i.e., a waiting thread is not always overtaken by other threads). Peterson’s algorithm, as well as ticket and bakery algorithms are fair

Algorithmic solutions to mutual exclusion

Dekker's and Peterson’s algorithms belong to the worst solutions to mutual exclusion. They require no support from the processor or the operating system. However, they rely on polling and have other requirements which can hardly be fulfilled in contemporary computers

Low-level mutual exclusion: TestAndSet

Instruction TestAndSet (TSL)

```
int TSL(int *lock) /* atomic */
{
  int old_lock_state = *lock;
  *lock = TRUE;
  return old_lock_state;
}

int lock = FALSE; /* initialisation */

while (TSL(&lock) == TRUE) /* enter critical section */
  ; /* or dequeue this thread */
critical section;
lock = FALSE; /* leave critical section */
```

Low-level mutual exclusion: CompareAndSwap

Instruction CompareAndSwap (CAS)

```
int CAS(int *lock, int old_state, int new_state) /* atomic */
{
  int old_lock_state = *lock;
  if (old_lock_state == old_state)
    *lock = new_state;
  return old_lock_state;
}

int lock = FALSE; /* initialisation */

while (CAS(lock, FALSE, TRUE)) /* enter critical section */
  ; /* or dequeue this thread */
critical section;
lock = FALSE; /* leave critical section */
```

Low-level mutual exclusion: futex

Futex (fast userspace mutex) is a data structure

used in Linux to implement the synchronization primitives of the pthread library

```
#include <linux/futex.h>
#include <sys/time.h>

int futex(int *uaddr, int futex_op, int val,
  const struct timespec *timeout,   /* or: uint32_t val2 */
  int *uaddr2, int val3);
```

The futex_op can be one of: **FUTEX_WAIT**, **FUTEX_WAKE**, FUTEX_FD, FUTEX_REQUEUE, FUTEX_CMP_REQUEUE, FUTEX_WAKE_OP, FUTEX_WAIT_BITSET, FUTEX_WAKE_BITSET

Low-level mutual exclusion: futex

```
/* Acquire the futex pointed to by 'futexp': wait for its value to
become 1, and then set the value to 0. */
static void fwait(int *futexp)
{
  int s;
  while (1)
  {
    /* Is the futex available? */
    if (CAS(futexp, 1, 0))
      break; /* Yes */
    /* Futex is not available; wait */
    s = futex(futexp, FUTEX_WAIT, 0, NULL, NULL, 0);
    if (s == -1 && errno != EAGAIN)
      errExit("futex-FUTEX_WAIT");
  }
```

Low-level mutual exclusion: futex

```
static void fpost(int *futexp)
{
  int s;
  if (CAS(futexp, 0, 1))
  {
    s = futex(futexp, FUTEX_WAKE, 1, NULL, NULL, 0);
    if (s == -1)
      errExit("futex-FUTEX_WAKE");
  }
}
```

Threads (POSIX threads, pthreads)

API (the core):

pthread_create()	starts a new thread, returns its id
pthread_self()	returns id of this thread
pthread_equal()	compares 2 ids
pthread_join()	waits for termination of a thread
pthread_mutex_init()	initialises a mutex
pthread_mutex_destroy()	deinitialises a mutex
pthread_mutex_lock()	locks mutex
pthread_mutex_unlock()	unlocks mutex
pthread_cond_init( )	initialises a conditional variable
pthread_cond_destroy( )	deinitialises a conditional variable
pthread_cond_signal( )	signals a conditional variable
pthread_cond_wait()	waits on a conditional variable

Threads (POSIX threads, pthreads)

Príklad: pthread_create a pthread_join

```
#include <pthread.h>
#include <stdio.h>

void *thread_routine(void* arg) {
        printf("Inside newly created thread\n");
}

int main() {
        pthread_t thread_id;
        void *thread_result;
        pthread_create( &thread_id, NULL, thread_routine, NULL );
        printf("Inside main thread \n");
        pthread_join(thread_id, &thread_result );
}
```

gcc test.c -lpthread

Threads (POSIX threads, pthreads)

The previous program ignores the return values of pthread_create() a pthread_join(). It does not matter on a slide, but it does matter on a computer:

```
if ((pthread_create(&thread_id, NULL, thread_routine, NULL) != 0)
{
    ERROR(“pthread_create() failed\n”);
}
```

Pthreads: producer-consumer

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int shared_data =1;
void *consumer(void* arg) {
      int i;
      for (i =0; i < 30; i ++ ) {
            pthread_mutex_lock( &mutex );
            shared_data --; /* Critical Section. */
            pthread_mutex_unlock( &mutex );
       }
}
void main() {
      int i;
      pthread_t thread_id;
      pthread_create(&thread_id, NULL, consumer, NULL);
       for(int i =0; i < 30; i ++ ) {
            pthread_mutex_lock( &mutex );
            shared_data ++; /* Producer Critical Section. */
            pthread_mutex_unlock( &mutex );
       }
       pthread_join(&thread_id);
       printf("End of main =%d\n”, shared_data);
}
```

Pthreads: producer-consumer (bounded buffer)

```
#define QUEUE_SIZE 10
#define NR_ITERATIONS 1000

int in = 0; /* reading index */
int out = 0; /* writing index */
int queue[QUEUE_SIZE];
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int waiting_empty = 0;
pthread_cond_t empty_cond = PTHREAD_COND_INITIALIZER;
int waiting_full = 0;
pthread_cond_t full_cond = PTHREAD_COND_INITIALIZER;

void main()
{
  pthread_t thread_consumer;
  pthread_create(&thread_consumer, NULL, consumer, NULL);
  producer();
  pthread_join(&thread_consumer);
}
```

Pthreads: producer-consumer (bounded buffer)

```
void *consumer(void* arg)
{
  int i;

  for (i = 0; i < NR_ITERATIONS; i++)
  {
    pthread_mutex_lock(&mutex); /* Begin critical section */
    if (in == out)
    { /* The queue is empty */
      waiting_empty++;
      do
        pthread_cond_wait(&empty_cond, &mutex);
      while (in == out); /* Why this loop? */
      waiting_empty--;
    }
    /* When we get here, the queue is not empty. Read from queue[in], increase in */
    in = (in + 1) % QUEUE_SIZE;
    if (waiting_full > 0) /* Why not just cond_signal? */
      pthread_cond_signal(&full_cond);
    pthread_mutex_unlock(&mutex); /* End critical section */
  }
}
```

Pthreads: producer-consumer (bounded buffer)

```
void *producer(void *arg)
{
  int i;

  for (i = 0; i < NR_ITERATIONS; i++)
  {
    pthread_mutex_lock(&mutex);      /* Begin critical section */
    if ((in + 1) % QUEUE_SIZE == out)
    { /* The queue is full */
      waiting_full++;
      do
        pthread_cond_wait(&full_cond, &mutex);
      while ((in + 1) % QUEUE_SIZE == out); /* Why this loop? */
      waiting_full--;
    }
    /* When we get here, the queue is not full. Write to queue[out], increase out */
    out = (out + 1) % QUEUE_SIZE;
    if (waiting_empty > 0) /* Why not just cond_signal? */
      pthread_cond_signal(&empty_cond);
    pthread_mutex_unlock(&mutex);  /* End critical section */
  }
}
```

Pthreads: programming discipline

- Shared data **must** always be accessed through a single mutex
- The thread which locks a mutex also unlocks the mutex
- A boolean condition (expressed in terms of program variables) is associated with each condition variable. Every time the value of this boolean condition changes, do not forget to signal() the condition variable
  - Call Signal with a locked mutex; and only call it when you are certain that a thread is waiting for the signal
- Beware of deadlocks

Semaphores and mutexes

- In theory, mutex = binary semaphore (they offer different methods but can be simulated with each other)
- In Unix systems, there is a huge difference between a semaphore and a mutex. Mutexes exist only in the scope of the process. Semaphores exist in the scope of the operating system

Pthreads: Linux (contention scope)

- Contention scope is the POSIX term for describing bound and unbound threads
  - A bound thread is said to have system contention scope, i.e. it contends with all threads in the system
  - An unbound thread has process contention scope, i.e. it contends with threads in the same process

Linux uses bound threads. But think of scheduling policies. You cannot influence scheduling of (all) processes, but you should be able to schedule (all) your threads within a process in any way you like. With Linux, you can not do that (and not only with Linux)

Pthreads: mutex_attr, cond_attr, ...

Pthreads offers many other functions and advertises to call them with other arguments besides those which shown in previous slides

Do not use use the other functions and arguments, you will make less mistakes. Theory of *concurrent programming* does not make use of them either and is still sufficiently interesting (far from trivial)

Pthreads: mutex_attr, cond_attr, ...

Example:

pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutex_attr *attr)

Correct use:

pthread_mutex_init(&mutex, **NULL**);

Incorrect use:

any other

NULL means default behaviour (fast mutex). An error checking mutex may be useful during debugging. But beware of *recursive mutex* which mimicks locking in Java, and which is never needed