Chapter 10

Threads

Threads

- **Threads are “light-weight processes”**
- **Threads are “processes inside a process”**
- **Threads share process data structures**
- **Threads need to synchronise**
- **Many synchronisation primitives are geared towards threads**
- **In C: thread libraries**
- **In Java: threads are part of the language**

OS Processes: Control and Resources

A process as managed by the operating system groups together:

- **Resources:** open files, global variables, child processes, signal handlers, accounting information, …
- **Control of Execution — “Thread of Control”:**

  program counter, registers, stack

Multithreaded Processes

One process may have **multiple threads of control**

- Each **thread** of a process has **its own control**:

  thread id, program counter, register set, stack, etc.
- The threads **share the resources** of the process:

  address space, file descriptors, …

Threads sometimes called **lightweight processes**

Benefits over cooperating processes:

- Resource sharing
- Economy

Kinds of Threads

User threads

- Invisible to the kernel; managed by a thread library in user space
- Threads of same process scheduled as a unit
- Lower cost

Kernel threads

- Created, scheduled, and managed by kernel
- Threads of the same process can be scheduled independently (possibly on different processors)
- Higher cost

Multithreading Models

- **Many-to-one:** Several user threads to one kernel thread
  - Threads of a process managed as a unit
  - Used for multithreading in absence of kernel threads
- **One-to-one:** One user thread to one kernel thread
  - Threads of a process managed independently
  - Can be costly because users can cause many kernel threads to be created
- **Many-to-many:** Several user threads to several kernel threads
  - More flexibility than many-to-one
  - Less costly than one-to-one
  - Used for multithreading on a multiprocessor

Thread Programming Styles

Different ways to organise multithreaded programs:

Reactive:

- Thread runs in an infinite loop
- Continuously checks for certain events to occur
- Responds to the events when they occur

Task oriented:

- Parent checks all relevant events
- Parent thread creates a child thread to do a task
- Parent is notified when task is completed
- Child terminates when task is completed

Multithreaded Servers

On-demand spawning: New thread for every request, terminates after request serviced.

- overhead of ***thread creation for every request***

Thread-pool: Spawning a “pool” of threads at startup; assign incoming requests to ***idle threads from pool***

- lower overhead per request
- dynamic adjustment of pool size possible

Apache 2 (Final 2002, 10%)

Version 2 of the WWW server Apache can work with different multiprocessing/multithreading arrangements.

Under UNIX, the default is a hybrid multi-process multi-threaded server. Each process has a fixed number of threads (usually 20). The server adjusts to handle load by dynamically increasing or decreasing the number of processes.

(a) Explain which advantages this arrangement has over pure multiprocessing (without multithreading).

(b) Explain which advantages this arrangement has over pure multithreading (without multiprocessing).

Thread Programming Interfaces

OS Level: programming via system calls or wrapper libraries

Library Level: may or may not be portable

Programming Language Level: Implementation in run-time system or mapped to OS threads or thread library

---

Spawning a New Thread

- Specify **main procedure** for new thread
- If necessary, specify arguments
- New **sibling** thread is spawned
- Thread-Id is returned

Thread Termination

Suppose, the “target thread” needs to be cancelled.

Asynchronous cancellation: Active thread immediately terminates target thread

— could block resources

Deferred cancellation: Target thread checks at cancellation points whether it should terminate

— could waste CPU

Forking and Signals in Multithreaded Processes

fork() duplicates only the calling thread:

- Other threads can “clean up” with *pthread_atfork*().
- “*forkall*()” could duplicate all threads — not in POSIX

exec() overlays **all** threads.

Choices for (UNIX) **signals:** Deliver signal to

- specific target thread
- every thread in process
- certain threads in the process
- designated signal handling thread

Threads vs. Processes

spawning	forking
sibling	child
joining	waiting
shared memory	copied memory
shared resources	copied resource access
various synchr. methods	signals, pipes

Thread Scheduling Issues

- User-level threads: user-level scheduling
  - Preemptive or non-preemptive
  - flexible priority systems possible
  - no parallelism on multiprocessors
- Kernel-level threads: OS scheduling, parallelism possible
- **Gang scheduling:** on multiprocessors, multiple threads belonging to one process are scheduled at the **same time** on **different processors**, enhancing fast communication.

Thread Programming Environments

Pthreads: POSIX 1003.1c

Solaris 2: User-level threads, **unbound** or **bound** to **lightweight processes** (LWPs)

Windows 2000: kernel-level threads **fiber** library for many-to-many mapping

Linux: `clone` system call opens spectrum between processes and threads

Java: language-level threads

Sections 5.4 – 5.8 in [Silberschatz] — **Read!**

POSIX Threads

- **Specification** of a thread package **interface**
- Implementations need not provide all features
- User-level and kernel-level (partial) implementations possible
- Some features (usually) require kernel-level support

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

USP: ***“Do not let the prototype of*** *pthread_create* ***intimidate you—threads are easy to create and use.”***

POSIX Thread Creation

```
#include <pthread.h>                          /* USP Example 12.4 */
#include <stdio.h>

void * processfd(void * arg);

int error, fd;
pthread_t tid;

if ((fd = open("my.dat", O_RDONLY)) == –1)
   perror("Failed to open my.dat");
else if (error = pthread_create(&tid, NULL, processfd, &fd))
   fprintf(stderr, "Failed to create thread: %s\n",
                  strerror(error));
else
   printf("Thread created\n");
```

POSIX Thread Attributes

- Thread attributes are organised in a (possibly shared) **thread attribute object** (see `man pthread_attr_init`):
  - **detached** or **joinable**
  - scheduling policy: real-time (FIFO or round-robin) or “other”
  - priority for real-time threads
  - “scheduling contention scope”: process (user-level) or system (kernel-level)
  - POSIX standard terminology is intentionally **OS independent!**
- Preemptive thread scheduling **not specified** — use *sched_yield*() (from sched.h) to guarantee fairness!

POSIX Threads: Detaching and Joining

- A thread can exit by calling *pthread_exit* or by returning from its start function.
- When a **detached** thread exits:
  - its resources are released,
  - its return value is never inspected.
- Non-detached threads can be “joined” (like *wait* for processes):
  - assume thread t_1 calls $join(t_2)$
  - thread t_1 blocks until t_2 terminates
  - return value of t_2 is available to t_1.
  - `int pthread_join(pthread_t thread, void ** value_ptr);`
  - Joinable threads must take care to return pointers valid beyond their own existence!

POSIX Threads: Detaching Examples

One thread can detach another thread:

```
if (error = pthread_create(&tid, NULL, processfd, &fd))
   fprintf(stderr, "Failed to create thread: %s\n",
                  strerror(error));
else if ( error = pthread_detach(tid) )
   fprintf(stderr, "Failed to detach thread: %s\n",
                  strerror(error));
```

A thread can detach itself:

```
void * detachfun( void * arg ) {
  int i = *((int *)(arg));
  if ( ! pthread_detach( pthread_self() ) )  return NULL;
  fprintf(stderr, "My argument is %d\n", i);
  return NULL;
}
```

POSIX Threads: Joining Examples

Retrieving the return value of thread *tid* :

```
int error;
int * exitcode;
if (error = pthread_join(tid, &exitcodep))
   fprintf(stderr, "Failed to join thread: %s\n",
                    strerror(error));
else
   fprintf(stderr, "The exit code was  %d\n", *exitcodep);
```

What happens in the following?

pthread_join(pthread_self());

— ***may*** return *EDEADLK*, if implementation detects deadlocks.

POSIX Thread Cancellation Settings

Cancellation state (*pthread_setcancelstate*):

- *PTHREAD_CANCEL_ENABLE*: default
- *PTHREAD_CANCEL_DISABLE*: ignores cancellation requests

Cancellation type (changed through *pthread_setcanceltype*):

- *PTHREAD_CANCEL_ASYNCHRONOUS*: cancellation requests are immediately honoured
- *PTHREAD_CANCEL_DEFERRED* (default): cancellation requests are kept pending until the next **cancellation point**: *pthread_join*, *pthread_cond_wait*, *pthread_cond_timedwait*, *pthread_testcancel*, *sem_wait*, *sigwait*

General rule: A function that changes cancellation state or type should restore the original settings before returning!

POSIX Thread Cancellation

`int pthread_cancel(pthread_t thread);`

sends a **cancellation request**.

Honoring a cancellation request is effectively like calling *pthread_exit*(*PTHREAD_CANCELED*).

`void pthread_exit(void *retval);`

terminates thread, performs cleanup and finalisation, and makes *retval* to joining thread.

- A stack of **cleanup handlers** can be maintained with *pthread_cleanup_push* and *pthread_cleanup_pop*
- **Finalisation functions** for **thread-specific data**, can be registered via *pthread_key_create*

Thread-Specific Data

- Two views of global variables:
  - **Resource:** should have one instance per process
  - **Part of control:** should have one instance per thread
    - This is not supported by regular global variables
    - Special mechanism: **thread-specific data**
- In POSIX: Each thread has a private memory block, the **TSD area**
- Essentially: array of void pointers, indexed by **key**s

```
int pthread_key_create(pthread_key_t *k, void (*destr)(void*));
int pthread_key_delete(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void * ptr);
void * pthread_getspecific(pthread_key_t key);
```

Thread Safety

A function is **thread-safe** if multiple threads can execute invocations that are active at the same time (i.e., have an activation record on the different thread stacks at the same time).

Library functions that are ***not thread-safe*** can produce **interference** between threads! ⇒ **spurious errors!**

- *getpwent*, *gethostbyname*, *dirname*, *readdir*, *rand*, …
- Access of shared data: *getenv*, …
  - — *errno* is usually a macro.

Thread-safe variants of unsafe functions: suffix “*_r*” for “re-entrant”.

Thread Safety — *errno*

```
#include <errno.h>                    /* errnotest.c */
#include <stdio.h>

int main() {
  printf("%d\n", errno);
  return 0;
}
```

Result of `gcc -E errnotest.c | tail -4`:

```
int main() {
  printf("%d\n", (*__errno_location ()) );
  return 0;
}
```

Thread Interference

```
#include <pthread.h>                           /* interfere.c */
#include <stdio.h>
#include <stdlib.h>  /* for strtol   */
#include <string.h>  /* for strerror */
static volatile long int counter = 0;
static long int max;

void * count(void * arg) {          /* main function for thread */
  long int i, reg, * mycounter = arg;
  for(i = 0; i < max; i++) {
    reg = *mycounter;
    reg = reg + 1;
    *mycounter = reg;
  }
  return NULL;
}
```

```
int main (int argc, char * argv[]) {
  int error;
  pthread_t tid;
  max = strtol(argv[1], NULL, 10);
  if (error = pthread_create(&tid, NULL, count, &counter))
   fprintf(stderr, "Failed to spawn: %s\n", strerror(error));
  else {
   count( &counter );
   if (error = pthread_join(tid, NULL))
     fprintf(stderr, "Failed to join: %s\n", strerror(error));
   else
     fprintf(stderr, "The final count was  %ld\n", counter);
  }
  return 0;
}
```

Process Synchronization — Background

- Concurrent access to **shared data** may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- **Race condition:** The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be **synchronized**.

Example Race Condition

Implementation of “++*counter*;”:

(a) $reg = counter;$
(b) $reg = reg + 1;$
(c) $counter = reg;$

Assume:

- The shared *counter* variable starts out as 1
- Two processes P_0 and P_1 execute this implementation of “++*counter*;” **concurrently** (with **different registers**)
- **What is the final value of *counter*?**

Answer:

- $(P_0 \,||\, P_1)$ has 20 different traces
- The final values for *counter* in all these interleavings:
  [3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3]
- *“If anything can go wrong, it will.”* — Murphy’s Law

Mutual Exclusion

An **intended atomic action** can be breached (**fundamental safety failure**) if processes (or threads) share data structures

Two approaches for preventing breached atomic actions, based on **mutual exclusion**:

- At most one process can be in a critical section at a given time
- At most one process can be modifying a shared data structure at a given time

These are special cases of **process synchronization**

The two main devices for implementation:

- Semaphores
- Monitors

Process (Thread) Synchronization

- **Process synchronization** is when one process waits for an event to occur in another process
- Two special cases of process synchronization are needed for mutual exclusion:
  - When one process waits for another to leave a critical section
  - When one process waits for another to give up control of a shared data structure

The Critical-Section Problem

- *n* processes all competing to use some shared data.
- Each process has a code segment, called **critical section**, in which the shared data is accessed.
- **Problem:** Ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Specification of Solution to Critical-Section Problem

A solution to the critical-section problem must satisfy three properties:

1. **Mutual Exclusion**
2. **Progress**
3. **Bounded Waiting**

Mutual Exclusion

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

Progress

If

– no process is executing in its critical section,

– and there exist some processes that wish to enter their critical section,

then the selection of the process that will enter the critical section next cannot be postponed indefinitely.

Bounded Waiting

A bound must exist on **the number of times** that other processes are allowed to enter their critical sections

- after a process has made a request to enter its critical section
- and before that request is granted.

Process speeds:

- Assume that each process executes at a **non-zero speed**
- **No assumption** concerning **relative speed** of the *n* processes.

Specification of Solution to Critical-Section Problem

1. **Mutual Exclusion:** If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress:** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting:** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j):

```
while (true) {
         entry section
         critical section
         exit section
         remainder section
}
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1

Shared data:

`int turn;`

- initially: `turn = 0`
- semantics: `turn == i` ⇒ P_i can enter its critical section

Process P_i:

```
while (true) {
      while (turn != i) {}
        critical section
        turn = j;
        remainder section
}
```

Satisfies **mutual exclusion**, but **not progress:** **Deadlock** on non-alternating access

Algorithm 2

Shared data:

`boolean flag[2];`

- initially: `flag[0] = flag[1] = false`
- `flag[i] = true` ⇒ P_i ready to enter its critical section

Process P_i:

```
while (true) {
      flag[i] := true;
      while (flag[j]) {}
      critical section
      flag[i] = false;
      remainder section
}
```

Satisfies **mutual exclusion**, but **not progress: Deadlock** on simultaneous requests.

Algorithm 3 [Peterson 1981]

Shared data: combined from algorithms 1 and 2

```
int turn;
boolean flag[2];
```

Process P_i:

```
while (true) {
         flag[i] := true;
         turn = j;
         while (flag[j] && turn == j) {}
         critical section
         flag[i] = false;
         remainder section
}
```

Meets all three requirements: **Solves the critical-section problem for two processes**

Synchronization Hardware: Test-and-Set

Machine instruction to **test and modify** the content of a word **atomically:**

```
boolean TestAndSet(boolean * target) {
   boolean result = *target;
   *target = true;
   return result;
}
```

“Pseudo-C”

Mutual Exclusion with Test-and-Set

Shared data:

`boolean lock = false;`

Process P_i:

```
while (true) {
       while (TestAndSet(&lock))
             {}
       critical section
       lock = false;
       remainder section
}
```

Synchronization Hardware — Swap

Machine instruction to **atomically** **swap** two variables:

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

“Pseudo-C”

Mutual Exclusion with Swap

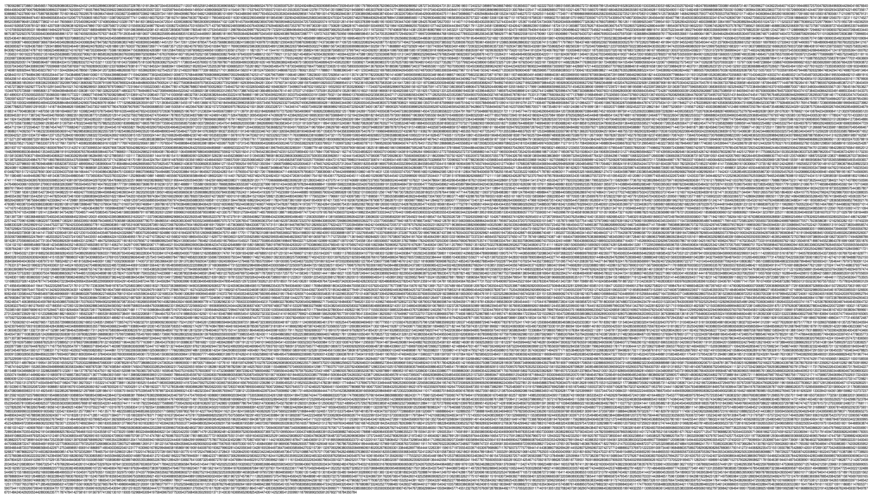
Shared data (initialized to false):

```
boolean lock;
```

Process P_i:

```
while (true) {
      key = true;
      while (key == true) Swap(&lock, &key);
      critical section
      lock = false;
      remainder section
}
```

Number of Interleavings for Two 100,000-Step Processes



Bakery Algorithm

Critical section for *n* processes:

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1, 2, 3, 3, 3, 3, 4, 5…
- Originally designed for distributed implementation [Lamport 1974]
- Simpler than the first known algorithm with $(n - 1)$ as waiting bound [Eisenberg, McGuire 1972]

Bakery Algorithm

- **Lexicographical order** on *ticket_number* × *process_id*:

  $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$

- *arrayMax* (*a*,*n*) returns a number k such that $k \geq a[i]$

  for all $i \in \{0, \ldots, n - 1\}$

- **Shared data:**

  boolean choosing[*n*]; /* initialise to false */
  long long int *number*[*n*]; /* initialise to 0 */
  /* **64 bit** may be okay for about **600 years** */

Bakery Algorithm

```
while (true) {
   choosing[i] = true;
   number[i] = arrayMax(number, n) + 1;
   choosing[i] = false;
   for (j = 0; j < n; j++) {
      while (choosing[j]) {}
      while (number[j] != 0 &&
        (number[j],j) < (number[i],i)) {}
   }
   critical section
   number[i] = 0;
   remainder section
}
```

Solves the critical-section problem for *n* processes.

---

n-Process Mutual Exclusion with Test-and-Set

The previous algorithms with `TestAndSet` and `Swap`

– can be used for *n* processes,

– but do not guarantee bounded waiting.

For guaranteeing bounded waiting, the `TestAndSet` lock can be used to additionally protect passing of “turns”:

Shared data:

```
boolean waiting[n];  /* all initialised */
boolean lock;        /* to false        */
```

Note: only Booleans!

---

n-Process Mutual Exclusion with Test-and-Set

```
while (true) {
    waiting[i] = true;
    boolean locked = true;
    while ( waiting[i] && locked ) locked = TestAndSet(&lock);
    waiting[i] = false;
    critical section
    j = (i+1) % n;
    while ( (j ≠ i) && !waiting[j] ) j = (j+1) % n;
    if (j == i) lock = false; else waiting[j] = false;
    remainder section
}
```

Elementary Solutions of the Critical-Section Problem

Necessary conditions satisfied: 1. **Mutual Exclusion**
2. **Progress**
3. **Bounded Waiting**

Solutions for 2 processes:

- **Software:** Peterson’s algorithm
- **Hardware:** Test-and-set instruction, swap instruction

Solutions for *n* processes:

- **Software:** Bakery algorithm — **unbounded counters**
- **Hardware:** needs only arrays of Booleans

All use busy waiting!

Semaphores

- The notion of a semaphore was invented by E.W. Dijkstra (*The Structure of the “THE”-Multiprogramming System*, 1968)
- Provide two services:
  - Mutual exclusion
  - Interprocess or interthread signaling
- Two basic kinds:
  - A **binary semaphore** serves as a resource access key
  - A **counting semaphore** serves as a resource availability measure
- Semaphores reduce the general problem of breached atomic actions to a simpler problem

---

Semaphores

- Synchronization tool that does not require busy waiting.
- Semaphore *S* — shared integer variable
- interface: *wait* and *signal* — ***specification***:

```
test(S):     if(S > 0) {S—; return true;}  /* auxiliary fct. */
             else      return false;

wait(S):     while( !test(S) ) {}

signal(S):   S++;
```

- *test*(*S*) and *signal*(*S*) must be **atomic**
- Can be used as implementation: ***busy waiting!***.

Use of Binary Semaphore

Solving the critical section problem for *n* processes:

- A 1-bit integer variable *S*, called a **binary semaphore** or **mutex**, is shared among the processes
  - *S* is initialized to 1
- Each critical section has the following form:

```
wait(S);
critical section
signal(S);
```

- **Invariants:**
  - $S \in \{0, 1\}$.
  - $S = 1$ iff the resource is free.

Semaphore Implementation with Waiting Set

- Semaphore has now two components:
  - An integer variable S
  - A waiting set W for the resources
- Implementation of the two operations:
  - *wait*(S): Decrement S. If $S < 0$, add calling process to the waiting set and suspend it.
  - *signal*(S): Increment S. If $S \leq 0$, choose a process from the waiting set and have it resume.
- *wait*(S) and *signal*(S) must be **atomic**
- **Invariant:**
  - If $S \leq 0$, then $|S| = |W|$.

Counting Semaphore

- A set of processes/threads sharing $N \geq 0$ **instances** of a resource
- A **counting semaphore**, is shared among the processes/threads; S is initialized to N.
- Each critical section has the following form:

```
wait(S);
critical section
signal(S);
```

- **Invariants:**
  - $S \leq N$.
  - If $S \geq 0$, there are S resources free.

Semaphore as a General Synchronization Tool

- **Problem:** Execute B in P_j only after A executed in P_i.
- **Solution:** Use semaphore `flag` initialized to `0`
- **Code:**

P_i	P_j
⋮	⋮
A	⋮
`signal(flag)`	`wait(flag)`
⋮	B

Counting Semaphores via Binary Semaphores

Data structures for counting semaphore S

```
binarySemaphore mutex;       /* initialized to 1 */
binarySemaphore nonempty;    /* initialized to 0 */
int counter:         /* initialized to initial value of S */
```

wait(S):

```
wait(mutex);
counter--;
if (counter < 0) {
   signal(mutex);
   wait(nonempty);
}
signal(mutex);
```

signal(S):

```
wait(mutex);
counter++;
if (counter <= 0)
   signal(nonempty);
else
   signal(mutex);
```

Semaphore Implementation Issues

- Semaphores are usually implemented in uniprocessor systems with noninterruptable system calls for *test* and *signal*
- Binary semaphores can be used to implement counting semaphores
- Semaphores ***guarding long critical sections*** ensure mutual exclusion via **their own, very short critical sections**.
- In multiprocessors, busy waiting for a short critical section is usually more efficient than context switching **— spinlocks**

Remarks on Semaphores

- The semaphore is a very simple and versatile concurrency control device
- Misapplied semaphores can lead to:
  - **Safety failure**: breakdown of mutual exclusion
  - **Liveness failure**: deadlock
- Semaphore programming errors may be very difficult to debug

---

Deadlock and Starvation

Deadlock: Let `S` and `Q` be two semaphores initialized to 1

P_i	P_j
⋮	⋮
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*);	*signal*(*S*);

Starvation: Indefinite blocking: A process may never be removed from the semaphore queue in which it is suspended.

Semaphores for POSIX Thread Synchronization

- **POSIX counting semaphores:**

```
#include <semaphore.h>
sem_t mysem;
sem_init(&mysem, 0, n) /* 0: process-local */
sem_wait(&mysem);
sem_post(&mysem);
```

- **PThread mutexes:** binary semaphores

```
#include <pthread.h>
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);
```

Protecting a Counter with a Mutex

```
#include <pthread.h>                                /* counter.c */
static int count = 0;
static pthread_mutex_t countlock =
PTHREAD_MUTEX_INITIALIZER;

int increment(void) {                   /* increment the counter */
  int error;
  if (error = pthread_mutex_lock(&countlock))
    return error;
  count++;
  return pthread_mutex_unlock(&countlock);
}
```

Protecting Unsafe Library Functions

```
#include <pthread.h>                               /* randsafe.c */
#include <stdlib.h>

int randsafe(double *ranp) {
  static pthread_mutex_t
     lock = PTHREAD_MUTEX_INITIALIZER;
  int error;

  if (error = pthread_mutex_lock(&lock))
    return error;
  *ranp = (rand() + 0.5)/(RAND_MAX + 1.0);
  return pthread_mutex_unlock(&lock);
}
```

Making Datastructures Thread-Safe

- Make each original access function static, i.e., **module-local**
- Add a **mutex** to each instance of the datastructures
- Produce a **wrapper** for each original function *f* that calls *f* only after ***acquiring all necessary mutexes***
- Consider **granularity of locking!**
- Example: Multiprocessor OS:
  - Lock the whole OS: only one CPU in kernel at any time: inefficient
  - Lock individual OS tables: smaller critical sections

At-Most-Once Execution

- Some initialisations **must not happen more than once**.
- Can be hard to do in other ways — special mechanism provided:

```
#include <pthread.h>                              /* (printinitonce.c) */
#include <stdio.h>

int var; /* exported variable, to be initialised only once */
static pthread_once_t var_initonce = PTHREAD_ONCE_INIT;

static void initialization(void)
 { var = 1;  printf("The variable was initialized to %d\n", var); }

int printinitonce(void) {     /* exported initialisation function */
  return pthread_once(&var_initonce, initialization);
}
```

At-Most-Once Execution — Different Approach

```
#include <pthread.h>                    /* printinitmutex.c */
#include <stdio.h>

int printinitmutex(int *var, int value) {
   static int done = 0;
   static pthread_mutex_t lock =
               PTHREAD_MUTEX_INITIALIZER;
   int error;
   if (error = pthread_mutex_lock(&lock)) return error;
   if (!done) {
      *var = value;
      printf("The variable was initialized to %d\n", value);
      done = 1;
   }
   return pthread_mutex_unlock(&lock);
}
```

At-Most-Once Execution — Remarks

- In both examples: It is **guaranteed** that assignment to *var* and message printing (*simple example*) are executed **at most once**
- USP motivation: POSIX mutexes must be initialised **only once**
- Most mutexes can be initialised at declaration:

  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
- This does not work for mutexes in *malloc*ed datastructures
- *pthread_mutex_t * locks = NULL*; /* shared variable */

  ...

  *locks = malloc(k ** sizeof(*pthread_mutex_t*));
- These mutexes have to be initialised using *pthread_mutex_init* — **exactly once!**

Advanced Synchronisation Mechanisms

If synchronisation primitives can be provided as language primitives, higher abstractions are possible:

- **Critical Regions** as language construct
- **Monitors:** built-in module or object locks
- **Condition variables:** more flexible suspending
  - normally part of monitor mechanism
  - in POSIX, used with mutexes instead

Language-Level Synchronization: Critical Regions

For processes with **explicitly shared** variables:

v : *shared* int

These shared variables can only be accessed in protected blocks:

region v when condition do *body*

While *body* is being executed, no other process can access *v*.

– Regions referring to *v* exclude each other in time.

– When a process tries to execute the region statement, *condition* is evaluated; if true, statement *body* is executed. Otherwise, the process is delayed until *condition* becomes true and no other process is in the region associated with *v*.

Critical regions can be implemented using semaphores.

Critical Regions Example: Bounded Buffer

Shared data: struct *buffer* { int *pool*[*n*];
int *count*, *in*, *out*; }

Producer process inserts *nextp* into the shared buffer:

```
region buffer when(count < n) {
  pool[in] ≔ nextp;
  in ≔ (in + 1) % n;
  count++;                    }
```

Consumer removes an item from the shared buffer and puts it in *nextc*:

```
region buffer when (count > 0) {
  nextc ≔ pool[out];
  out ≔ (out + 1) % n;
  count−−;                     }
```

Language-Level Synchronization: Monitors

- A **monitor** is a ***module*** or ***object*** with a mutual exclusion lock
  - A thread must obtain the lock on the monitor before it can execute one of the interface functions
  - The lock is released when the function is exited
  - Consequently, only one thread can be executing interface functions at a time
- Monitors are usually provided as a programming language construct
- A monitor can contain **condition variables** which are special variables used for coordinating access to the monitor
  - "Condition variables" ***do not contain any memory***

Why "Condition Variables"

Usage:

- A thread t_1 tests whether some **condition** c (i.e., a predicate) holds
- If false, t_1 waits
- If another thread t_2 changes any of the variables involved in c, this **might** make c true if tested again
- Therefore, t_2 signals to t_1 that testing the condition again makes sense

– A "condition variable" contains **no memory** for values

– A "condition variable" **does not imply any condition**

– It is only a **synchronization mechanism**, usually implemented by a ***waiting set***

Condition Variables: Wait and Signal

- Condition variables are used by calling two special operations on them: *wait* and *signal*
- If a thread calls *wait* on a condition variable x, then the thread
  - is suspended until another thread calls *signal* on x, and
  - gives up the lock on the monitor
- If a thread calls *signal* on a condition variable x, then one thread waiting on x is resumed
- Which thread is resumed after *signal* is called depends on the implementation
  - Signal-and-wait approach: signalling thread waits
  - Signal-and-continue approach: signalling thread continues

Condition Variables versus Semaphores

	Semaphore	**Condition Variable**
`wait`	decrements if positive waits only if ≤ 0	waits always
`signal`	increments if no waiting wakes up one waiting	no-op if no waiting wakes up one waiting

---

POSIX Condition Variables

- Are declared independent of any mutex
- Each condition variable **must** be used together with **always the same** mutex

  — programmer responsibility!

```
pthread_mutex_lock(&m);
while ( x ≠ y )
  pthread_cond_wait(&v, &m);
/* modify
   x or y
   if necessary */
pthread_mutex_unlock(&m);
```

```
pthread_mutex_lock(&m);
x++;
pthread_cond_signal(&v);
pthread_mutex_unlock(&m);
```

```
pthread_mutex_lock(&m);
y—;
pthread_mutex_unlock(&m);
pthread_cond_signal(&v);
```

Rules for Using Condition Variables (USP p.471)

- Acquire the mutex before testing the predicate
- Retest the predicate after *pthread_cond_wait*
- Acquire the mutex before changing any variables appearing in the predicate
- **Signal after changing any variables appearing in the predicate**
- Hold the mutex only for a short period of time
- Release the mutex either explicitly (*pthread_mutex_unlock*) or implicitly (*pthread_cond_wait*).
- **Think "monitor"!**
- **Think "**region *v* when *cond* do *body***"!**

POSIX Condition Variables

- Are declared independent of any mutex
- Each condition variable **must** be used together with **always the same** mutex

  — programmer responsibility!

```
pthread_mutex_lock(&m);
while ( x ≠ y )
  pthread_cond_wait(&v, &m);
/* modify
   x or y
   if necessary */
pthread_mutex_unlock(&m);
```

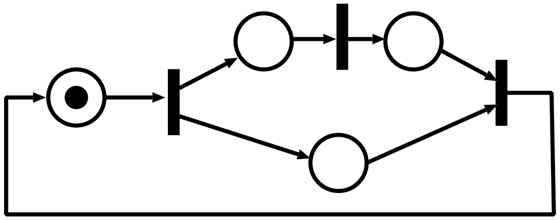
```
pthread_mutex_lock(&m);
x++;
pthread_cond_signal(&v);
pthread_mutex_unlock(&m);
```

```
pthread_mutex_lock(&m);
y—;
pthread_mutex_unlock(&m);
pthread_cond_signal(&v);
```

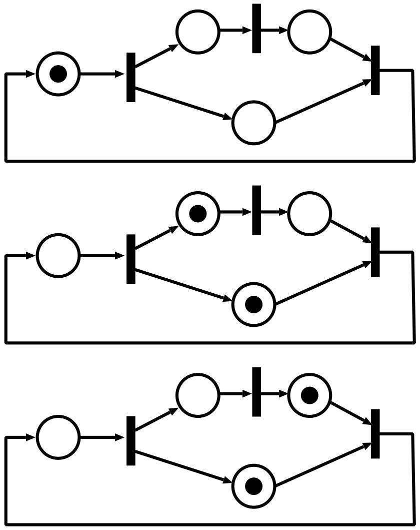
Another Synchronisation Mechanism: Barriers

A barrier is a **synchronisation point:**

- no process (or thread) passes the barrier before every other process has arrived at the barrier, too
- ***Alternative:*** only some fixed number of processes is required to pass the barrier
- Related formalism: **Petri nets**:



Petri Nets: The Token Game



A Thread Barrier Using POSIX Condition Variables

```
#include <errno.h>                              /* tbarrier.c */
#include <pthread.h>
static pthread_cond_t bcond = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t bmutex = PTHREAD_MUTEX_INITIALIZER;
static int count = 0;
static int limit = 0;
int initbarrier(int n) {    /* initialize the barrier to be size n */
   int error;
   if (error = pthread_mutex_lock(&bmutex))
      return error;          /* couldn’t lock, give up */
   if (limit ≠ 0)        /* barrier can only be initialized once */
    { pthread_mutex_unlock(&bmutex); return EINVAL; }
   limit = n;
   return pthread_mutex_unlock(&bmutex);
}
```

```
/* wait at the barrier until all threads arrive */
int waitbarrier(void) {
   int error, berror = 0;
   if (error = pthread_mutex_lock(&bmutex))
      return error;                 /* couldn’t lock, give up */
   if (limit ≤ 0)               /* make sure barrier initialized */
    { pthread_mutex_unlock(&bmutex); return EINVAL; }
   count++;
   while ((count < limit) && !berror)
      berror = pthread_cond_wait(&bcond, &bmutex);
   if (!berror)
       /* wake up everyone */
       berror = pthread_cond_broadcast(&bcond);
   error = pthread_mutex_unlock(&bmutex);
   if (berror) return berror; else return error;
}
```

Java Threads

Java threads may be created by:

- Extending the *Thread* class:
  - overriding the *run* method, and
  - invoking the *start* method of an object of that class

    start creates new thread running *run*.
- Implementing the *Runnable* interface:
  - defining the *run* method,
  - creating a new *Thread* object with an object of that *Runnable* class as constructor argument, and
  - invoking the *start* method of that *Thread* object.

Java threads are **managed by the JVM**.

Java Synchronization

- Objects are a kind of monitor
  - Each object has a mutual exclusion lock
  - A thread must obtain the lock before it can execute a **synchronized** method or block of code
  - Unsynchronized methods can be executed at any time
- Each object has **one unnamed condition variable**, a wait set, and wait and signal methods
  - Wait methods: three versions of *wait*
  - Signal methods: *notify* and *notifyAll*
  - *wait*, *notify*, and *notifyAll* can only be called from within synchronized methods or blocks
  - Signal-and-continue approach is used, but it is not specified which thread in the waiting set is resumed

---

synchronized Methods and Blocks

For methods, synchronization on this:

```
class C {
  synchronized int m(...) {...}
}
```

Synchronization on arbitrary objects:

```
class C {
  public int m(..) {
    ...
    synchronized(lock) {
      ...
    }
    ...
  }
}
```

“Global” Locks in Java

For making sure that every thread locks on the same object, one can use:

- locks on *Class* objects
- final static lock objects

Example:

```
class C {
  private final static Object LOCK = new Object ();
  private static long objCount = 0;
  private int _field;
  public C(int n) {
    synchronized(LOCK) { objCount++; }
    _field = n;
  }
```

notify() and *wait*()

- *obj.wait*() causes current thread to wait until another thread invokes a *notify* method for *obj*
  - can only be called inside a block synchronized on *obj*
  - releases lock on *obj*
  - can only continue after lock on *obj* has been re-acquired
  - waiting can be interrupted, causing *InterruptedException*
  - overloaded variants *wait*(...) allow to specify **time-out**
  - implemented using wait set
- *obj.notifyAll*() causes all threads in the wait set of *obj* to be runnable again
  - calling thread still has lock on *obj* and continues
  - awakened threads compete for lock on *obj*
- *obj.notify*() wakes up only one thread
  - should only be used if this is known to be safe!

Bounded Buffer with Monitors 1

Adopted from Silberschatz et al.

```
public class BoundedBuffer {
                                                   // Fields
 private Object[] _buffer;
 private int _count;
 private int _in, _out;
 private static final int BUFFER_SIZE = 10;

 public BoundedBuffer() {                          // Constructor
  _buffer = new Object[BUFFER_SIZE];
  _count = 0;
  _in = 0;
  _out = 0;
 }
```

Bounded Buffer with Monitors 2

```
public synchronized void enter(Object item) {
  while (_count == BUFFER_SIZE) {
   try {
    this.wait();
   }
   catch (InterruptedException e) { }
  }

  _count = _count + 1;
  _buffer[_in] = item;
  _in = (_in + 1) % BUFFER_SIZE;

  this.notifyAll();
 }
```

Bounded Buffer with Monitors 3

```
public synchronized Object remove() {
  Object item;
  while (_count == 0) {
   try { this.wait(); }
   catch (InterruptedException e) { }
  }

  _count = _count - 1;
  item = _buffer[_out];
  _out = (_out + 1) % BUFFER_SIZE;

  this.notifyAll();
  return item;
 }}
```

Well-disciplined Java Objects

A Java object is **well-disciplined** if the following conditions are satisfied:

- **Specification**: There is an intended **invariant** for the object
- **Correctness**:
  - All fields are initialized so that the object satisfies its invariant
  - All non-private methods preserve the object's invariant
- **Liveness**: All non-private methods terminate (if they are called when the object satisfies its invariant)
- **Safety**:
  - All fields are nonpublic
  - All non-private methods are synchronized