Chapter 12

Summary

Concurrency

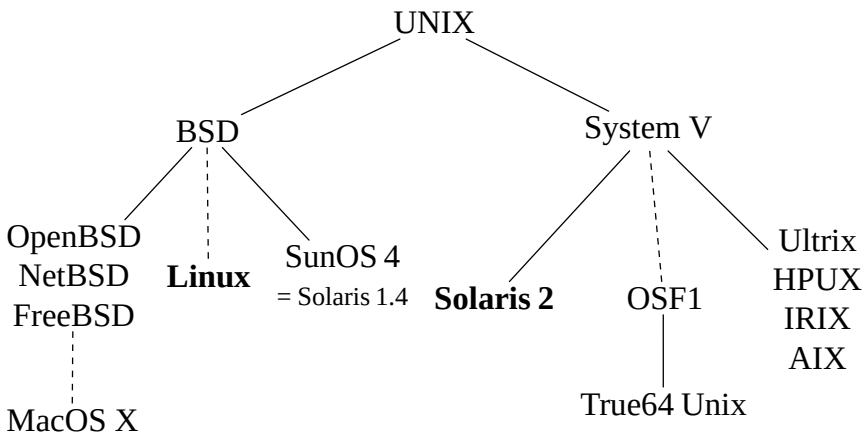
A **concurrent system** consists of a set (with at least two elements) of **threads of control** or **processes** that

- **execute** (essentially) **independently**
- may access **common resources**
- may **communicate** with each other

Operating System Components

- Process management
- Main memory management
- File management
- I/O management
- Secondary storage management
- Networking
- Security
- Command Interpreter

UNIX Flavours



POSIX

Bash Summary

- Complex language
- Context-sensitive lexing
- Complete imperative control structures
- Mostly dynamic binding (static binding with local)
- Iterated expansion mechanisms — functional flavour
- Concise syntax for command-line interaction
- Shell scripts **need documentation!**
- Shell scripts **need robustness!**
- Shell scripts **need security awareness!**

`man bash`

- **Options:** shell mode, and `set` options
- **Invocation:** shell modes, initialisation, `sh` emulation
- **Shell Grammar:** simple commands, pipelines, lists, compound commands, function definitions
- **Quoting:** special characters in different contexts
- **Parameters:**
  - positional parameters (arguments)
  - special parameters `$[*@#?-$0_]`
  - shell variables `$BASH*`, `$HOSTNAME`, `$PWD`, `$HOME`, `$PS1`
  - user-defined variables
  - array variables
- **Expansion:** ***"seven kinds"!***

`man bash` (ctd.)

- **Redirection:** *command args* < *infile* > *outfile* 2>> *errfile*
- **Aliases:** "The rules […] are somewhat confusing".

  "For almost every purpose, aliases are superseded by shell functions."
- **Functions:** local variables need to be declared **local**
- **Arithmetic evaluation, conditional expressions**
- **Simple command expansion, command execution environment**
- **Signals, job control**
- **Prompting, readline, history, history expansion**
- **Shell builtin commands**

Expansion

- **Brace expansion:** `a{d,c,b}e` → `ade ace abe`
- **Tilde expansion:** Home directories `~user` and `~`
- **Parameter and variable expansion:** `$varname`
  with default: `$varname:-default`
  various string manipulations possible
- **Command substitution:** `$(command)` or `‘command‘`
- **Arithmetic expansion:** `$((expression))`
- **Process substitution:** (obtaining FIFOs for processes)
- **Word splitting:** results of parameter expansion, command substitution, and arithmetic expansion are split into words (outside `"..."`)
- **Pathname expansion (globbing):** `dir*/file?.[coh]`
- **Quote Removal**

Disk Organization

- One **boot control block** per disk — information for booting an OS from that disk.
- Several **partitions**:
  - **partition control block**: partition size, block size, block management data structures (free block count, free bolck pointer, free FCB count, free FCB pointer)
  - **directory structure**
  - **File Control Blocks (FCBs) — in UNIX: inode:**
    - Ownership, premission information
    - Location of data blocks

inodes in Detail

- mode (permissions)
- number of hard links
- owner, group
- timestamps: modification mtime, access atime, change ctime
- size (in bytes)
- number of blocks allocated
- Pointers to allocated blocks:
  - 12 direct block pointers to the first data blocks of the file
  - one **single indirect** pointer, points to a block containing pointers to the next (blocksize / pointersize) data blocks
  - one **double indirect** pointer
  - one **triple indirect** pointer

File Tables in UNIX

In-memory Inode Table

- For each open file **a single copy** of its file control blocks
- Acts as **cache** for file control blocks

System File Table

- one entry per active *open*
- contains reference to in-memory inode
- current position, access rights, access mode

File Descriptor Table

- one per process — copied by *fork*
- contains reference to system file table entry
- In user space, references to file descriptors are int indexes into this table.
- file locks — not copied by *fork*

Building Software Packages From Source

- source-code only
  — look for main module, config.h; read; compile…
- README-based
  — **read** README, INSTALL; follow instructions
- Makefile-based
  - edit configuration part of Makefile
  - make all
  - make install (as root)
- configure-based (GNU auto-tools)
  - ./configure --help — notice enable* and with* options
  - ./configure --prefix=*/usr/local/packages/XYZ*
  - make — sometimes separately: make doc html ps
  - make install (as root)

Process Management Overview

- A **process** is a program in execution
- The CPU is **shared** among the processes **ready to run**
  - Only one process can run at a time
  - The CPU is rapidly switched between processes — **"context switch"**
- Processes communicate with the OS using **system calls**
- A process **can be interrupted at any time** by a device interrupt or a system call
- A process is represented by a data structure called a **process control block (PCB)** or a **process descriptor**

Context Switching

A **context switch** saves the state of the current running process and then loads the state of the next process to be executed.

It needs to save enough information about the current running process so that it can be resumed later *as if nothing had happened*. This determines important components of the PCB.

A context switch is initiated by an interrupt:

- **software interrupt** (system call)
- **device interrupt**
- **timer interrupt** (quantum expired)

Context Switch Steps

(Assuming a user process is interrupted)

- CPU senses interrupt
- CPU starts the interrupt handler **in privileged mode**
- *(Interrupt handler may disable all interrupts)*
- Interrupt handler stores state of interrupted process
- Interrupt handler executes (kernel) code for the interrupt
- Interrupt handler ends by calling the CPU scheduler
- CPU scheduler loads the state of the process it selects
- *(Interrupt handler would re-enable interrupts)*
- CPU switches back to **user mode** and executes selected process

Process Scheduling

- All processes are on the **job queue / process table**
  - Some *ready* processes may be swapped to disk.
  - Processes *in memory* that are ready to execute are also on the **ready queue**
  - Processes waiting for an I/O device are also on its **device queue**
- The **job scheduler** selects processes from the job queue to load into and out of memory
- When the CPU is free, the **CPU scheduler** selects one process from the ready queue to be executed by the CPU
  - A process can run until a timer interrupt or some other interrupt occurs
  - The CPU scheduler is called after the processing of each interrupt

Unix-Style Process Creation

Process creation in Unix involves four system calls:

- **fork**: Creates an exact copy of the calling process
  - Returns process id of child to parent
  - Returns **0** to child
- **execve**: Overwrites the calling process's program
- **exit**: Causes the calling process to terminate
  - Returns process id to waiting processes
- **wait**: Causes the calling process to wait until one of its children exits
  - Returns process id of exited child to process

Interruption by Signals

- **Signals** are a special kind of IPC "messages"
- A process can install **signal handlers**
- (Most) unhandled signals terminate the process
- Signals can arrive at any time
- Signals can interrupt certain system calls — *EINTR*

```
#include <errno.h>                    /* r_wait.c */
#include <sys/wait.h>

pid_t r_wait(int *stat_loc) {
  int retval;

  while (((retval = wait(stat_loc)) == -1) && (errno == EINTR)) ;
  return retval;
}
```


Signal Lifecycle, Long Version

1. A signal is **generated** and directed to a process
2. If the process **ignores** this signal, the signal is discarded
   — KILL, STOP cannot be ignored
3. If the process **blocks** this signal, it is kept **pending** until the process unblocks it
   — KILL, STOP cannot be blocked
4. Otherwise, the signal is **delivered** to the process
5. Once delivered, the signal must be **handled**
   - The **default** action is to ***terminate*** the process;
     - the process is ***stopped*** by STOP (^Z), TTIN, TTOU, TSTP (^S — type ^Q to continue)
     - CONT ***continues*** the process; CHLD is ignored
   - The default action is not taken if a signal handler has been installed for **catching** the signal
     — KILL, STOP cannot be caught

Pipes

- Pipes are kernel data structures for inter-process communication
- int *pipe*(int *filedes*[2]);
- ***Linux:*** *pipe* creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. *filedes*[0] is for reading, *filedes*[1] is for writing.
- "The ***POSIX*** standard does not specify what happens if a process tries to write to *filedes*[0] or read from *filedes*[1]."
- ***Solaris:*** The *pipe*() function creates an I/O mechanism called a pipe and returns two file descriptors, *fildes*[0] and *fildes*[1]. The files associated with *fildes*[0] and *fildes*[1] are streams and are both opened for reading and writing.

Named Pipes (FIFOs)

- Named pipes (FIFOs) are pipes turned into file system objects.
- A named pipe is a *special file* with access regulated via file system permissions:

```
prw-------  1 kahl users  0 Jan 30 00:08 /tmp/fifo1
```

- Data is passed though the FIFO by the kernel without writing it to the file system.
- Normally, opening the FIFO blocks until the other end is opened also.
- When a process tries to write to a FIFO that is not opened for read on the other side, the process is sent a *SIGPIPE* signal.

Avoiding Suspension on Individual *read* and *write* Calls

Problem:

- Normal *read* and *write* **block** until I/O possible
- Program may need to do other things while I/O impossible
- Program may need perform I/O where it **first** becomes possible

Different solutions:

- Open with *O_NONBLOCK* and "**poll manually**"
- Use *select*
- Use *poll*
- Open with *O_ASYNC* and perform I/O in signal handlers
- Use multiple threads (carefully …)

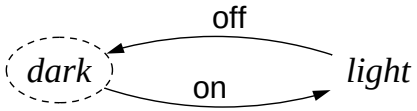
Labelled Transition Systems (LTSs)

Definition: A **labelled transition system** (S, s_0, L, δ) consists of

- a set S of *states*
- an *initial state* $s_0 : S$
- a set L of *action labels*
- a *transition relation* $\delta : \mathbb{P}(S \times L \times S)$.

Example:

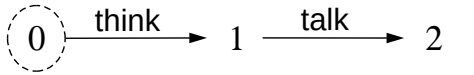
$$LightSwitch_1 = (\{dark, light\}, dark, \{\mathsf{on}, \mathsf{off}\}, \{(dark, \mathsf{on}, light), (light, \mathsf{off}, dark)\})$$



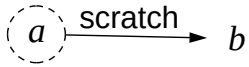
Concurrent Composition

- A system composed of several processes has a state that is composed from the states of the individual processes.

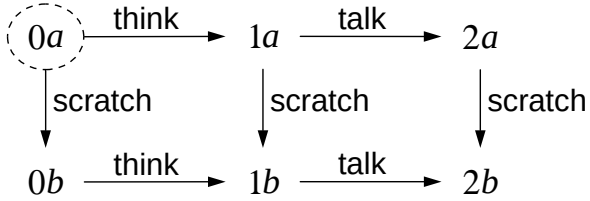
Converse =



Itch =



Converse || *Itch* =



While *Converse* and *Itch* have only one trace each, their composition has three, representing *arbitrary interleaving.*

Liveness and Safety Properties

A **safety property** asserts:

"something **bad** will **never** happen"

A **liveness property** asserts:

"something **good** will **eventually** happen"

Branching Transitions

A state of a process from which several transitions exist usually models one of the following:

- In this state, the process is prepared to **react** to different environmental stimuli
- In this state, the process **acts** by making a (non-deterministic) choice
  - non-determinism could be intended
  - non-determinism could be the result of abstraction

LTSs do not differentiate between **action** and **reaction**!

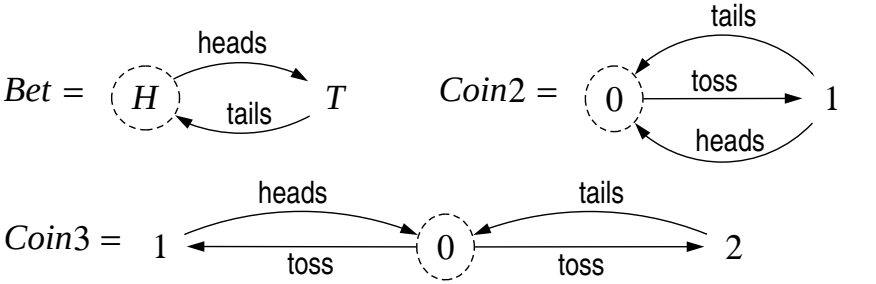
Non-Deterministic Choice, Traces, and Composition

*Coin*2 and *Coin*3 have the same trace set!

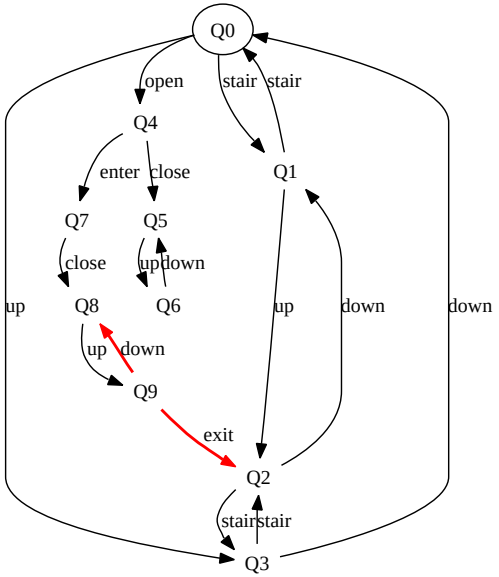
But, *Coin*2 || *Bet* and *Coin*3 || *Bet* have **different** trace sets!

$\Rightarrow$ Two LTSs P_1 and P_2 are **equivalent** iff for every LTS Q, the compositions $P_1 \parallel Q$ and $P_2 \parallel Q$ have the same trace set.

This is a *black-box* view: "No context enables distinction."



Fairness



Fairness assumption:

If a choice is arrived at infinitely often, then all of its branches are taken infinitely often.

Assuming fairness, additional lifeness properties hold, e.g.: "After an enter, there will eventually be an exit."

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**

Threads vs. Processes

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

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

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**.

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

Data Exclusion

- **Data exclusion** is the act of preventing access to data
  - Opposite of **data sharing**
- Examples of data access that needs to be prevented:
  - Hidden side effects
  - Violations of data invariants
  - Breached atomic actions
  - Unauthorized accesses

Data Exclusion Approaches

- Immutable data structures

  — *you **cannot change** the data*

- Side-effect-free sub-programs

  — *you **will not change** the data*

- Data confinement

  — *you* **cannot get at** *the data*

- Mutual exclusion

  — *you* **cannot interfere** *over the data*

Immutable Data Structures

- A data structure is **immutable** if its state cannot be changed
- Benefits:
  - Local assumptions satisfied by an immutable data structure are invariants of the data structure
  - Immutable data structures can be safely shared
- State changes realized by creating new immutable structures
- Two kinds of immutable objects:
  - Stateless objects having no fields whatsoever
  - Objects having only "final" fields
- Fully (and partially) immutable data structures should be used as much as possible
  - Values in an abstract data type should be immutable

Data Confinement

- **Confinement** can be used to ensure that only one thread can access a given data structure at a time
  - Similar to security measures used to ensure information privacy
  - Analysis of data flow is crucial
- Mechanisms for enforcing data confinement:
  - **Information hiding**: Interface is public; implementation is private
  - **Lexical scoping**: References to data structures are only visible in restricted portions of code
  - **Access control**: permissions for object access required
  - **Calling sequence**: Procedures are called in an order that excludes certain accesses to data

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

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);
```

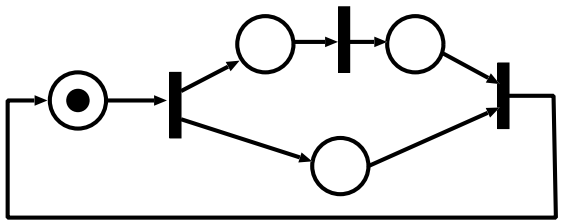
Condition Variables versus Semaphores

	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

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**:



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

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!

Method-Based Confinement Techniques

A method creates an object and then uses one of the following techniques to confine it:

- **Local confinement**: The object's reference is not allowed to leave the scope of the method
- **Tail call hand-off**: The object's reference is handed off in a tail call to another method
- **Call by value**: The object's *value* is given in a call to another method, i.e.,
  - a copy of the object is passed, or
  - information needed to reconstruct the object is passed.
- **Trust**: The object's reference is given in a call to another method which is trusted not to modify the object or forward its reference.

Thread-Based Confinement Techniques

- **Thread confinement**: All fields accessible within a thread are confined to the thread:
  - The thread behaves like a process with its own address space.
  - Supported by *java.lang.ThreadLocal*.
- **Thread hand-off**: The object references of a thread T are handed off to another thread T' in a tail call in T's run method.
- **Current thread confinement**: All thread object methods access the fields of the current thread object.

Object-Based Confinement Techniques

- **Adapters**: An object with unsynchronized methods is wrapped with an object with synchronized methods
- **Subclassing**: Unsynchronized methods of a class are synchronized in a subclass
- **Transfer protocol**: A resource object is passed between a group of objects according to a protocol
  - Example: Token ring

Summary

- **Shell Programming**
- **File Management**
- **UNIX Processes**: *fork*, *wait*, *exec*
- **POSIX signals**: synchronous and asynchronous, handling
- **IPC via (named) pipes**
- **Concurrency modelling using LTSs**
- Implementing concurrency inside processes: **Threads**
- **Process and thread synchronisation**
  - **Critical sections** — mutual exclusion
  - **Semaphors** and **monitors** — waiting for a simple lock
  - **Condition variables** — waiting for satisfaction of a complex condition
- Scheduling

Goals

- Know **what to expect** from a computer
- Know **what *not* to expect** from a computer
- Know **what can go wrong**, and **why**
- Know **what the public expects from your software**
- Know **how to achieve what you need**

Interaction with a Computer

Different interaction paradigms:

- **Point-and-click**
  - ***High* intuitivity** (sometimes)
  - ***Low* expressivity** and **abstraction** capabilities
- **Command line — *linguistic***
  - ***High* expressivity, abstraction** capabilities
  - ***High* intuitivity** — once you know the language

Become A Power-User

- Use a **Real Editor:**
  - Work through the Emacs tutorial
  - Learn the basics of `vi`; check out `vim`
  - Check whether you like Emacs or XEmacs better
- Improve your **command-line skills!**
  - Find out how to define shell functions and aliases
  - Do it!
- **Document preparation:**
  - not only WYSIWIG, i.e., ***"what you see is all you get"***
  - Check out LaTeX, Lout, and DocBook
- Read about **Literate Programming**
  - Try out FunnelWeb or NoWeb

Welcome to the World of Free Software

- Install some free software **from source**
- Find out about the **tools** involved in the process
- Write complete, user-friendly **makefiles** for some project
- Read about GNU and open source licences
- Find out about the differences between GNU/Linux, GNU/Hurd, FreeBSD, NetBSD, OpenBSD, Darwin, Solaris
- Check out some **technical** mailing lists or USENET news groups
- See how problems can be attacked — ***attack your own!***

More Skills

- Read the Documentation!
- Read the Questions!