Processes and Concurrency

Lecture on

Processes and Concurrency

Preserving speed and consistency

Walter Kriha

Goals

- Understand how the resource CPU is shared and managed
- Understand the importance of concurrent programming as THE means to achieve throughput and good response times.
- Understand the problems of concurrent access: data inconsistency and deadlocks
- Understand how Java handles concurrency and how threads can be supported by the operating system
- Learn to use an advanced threadpool design

It used to be system programmers which had to deal with complicated concurrency problems. Now it is application programmers running multiple threads for performance. We will look at the advantages and pitfalls of concurrent programming

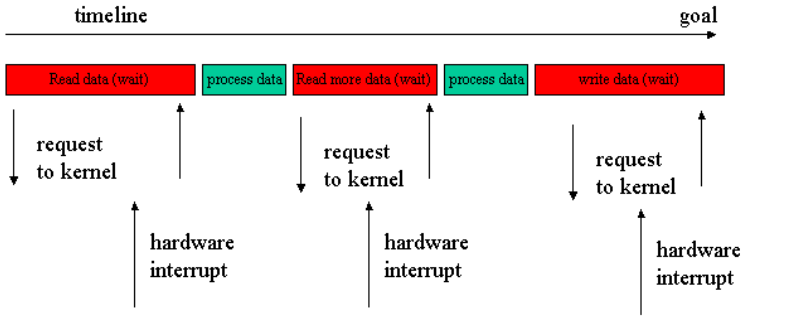
Procedure

- The basics of sharing CPUs (time sharing, context switches etc.)
- The process abstraction: sequentially progressing tasks
- process states and scheduling
- The thread abstraction: kernel and user threads
- synchronization primitives (mutex, semaphores, monitors)
- Inter-process communication

Getting the maximum out of threads requires a good understanding of how the specific OS implements them.

Processes

Why Processes?



A process is a representation of a sequential task. Asynchronous events like hardware interrupts from drives or networks are hidden behind a synchronous interface which blocks a process if it has to wait for data. The idea of sequential progress towards some goal is inherent to processes and at the same time something that programmers can understand (and program). Other forms of task organization are e.g. table driven state machines (finite-state machines) which are extremely hard to understand for humans. That is why e.g. chip design requires case and simulation tools.

What is a process?

an execution flow

- CPU register (instruction pointer etc.)
- page table entries
- cached instructions

a border around a set of resources

- file descriptors
- network connections
- shared memory regions
- rights
- user
- identity

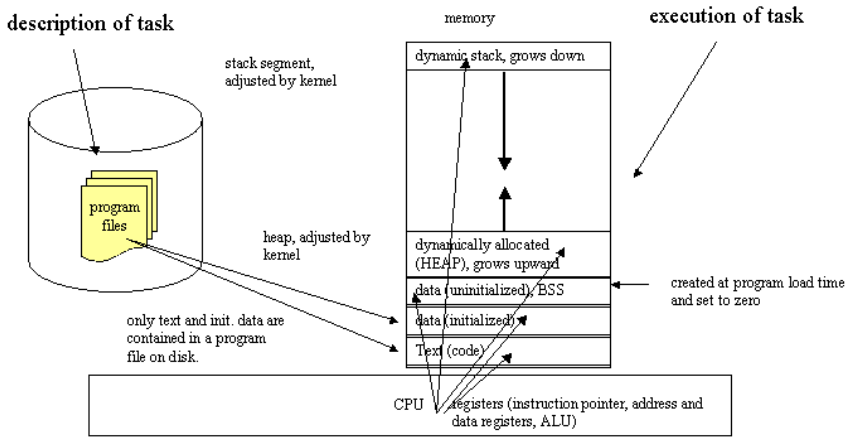
It is useful to separate the execution part from the resources. This allows us later to define a thread as an execution path over an existing set of resources. All in all is a process a very heavy-weight thing consisting of many data structures which must exist to make a process runnable. This in turn makes changing processes (context switching) an expensive operation.

Process Identity and Process Table

- Process ID
- Process state
- Registers, stack and instruction pointer
- Priority
- Runtime statistics (CPU time, wait time etc.)
- Waiting for which events
- Parent/child relations (if applicable)
- Memory segment (text, data, stack) pointers
- File descriptors, working directory, User and group ID

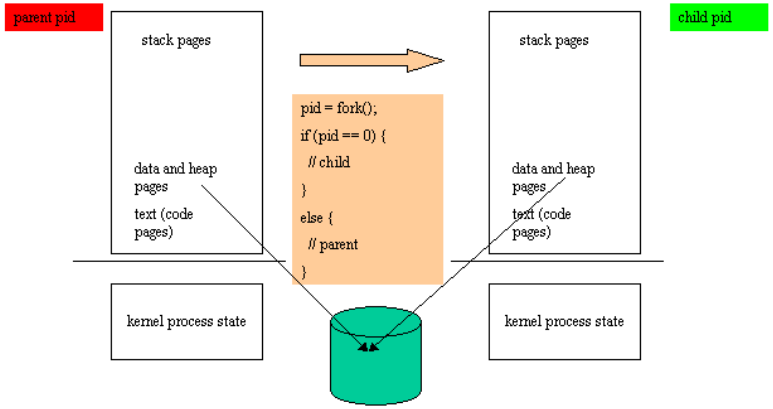
Just like files (inode table) or pages (page table entry) every process has a unique identity (PID). It is used to associate resources with it, kill the process or gather statistics for it.

From Program to Process



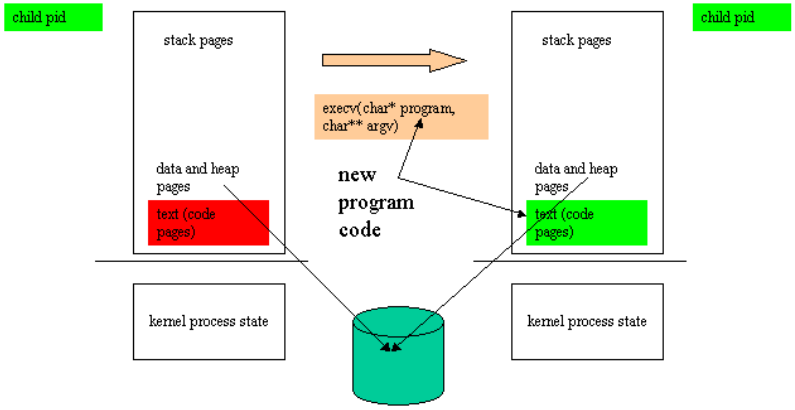
When a program is loaded into memory, three segments (text, data, stack) are created in memory and the address of the startup location is put into the instruction register of the CPU. The 3 segments together with the contents of the CPU registers form part of the state of the process. Without an operating system one program/process would be allocating the CPU resource completely.

How Process Creation Works: fork()



Usually a new process (child) is created by an existing process (parent) which uses the fork() system call to create an exact copy of the parent with a new process ID (pid). The new child process shares all memory segments with the parent and all resource handles (file descriptors to disks or network connections etc.). The return value of fork() differs for child and parent and allows the child to do different computations, e.g. to load a new program and execute it.

How Process Creation Works: exec()



When a child calls exec(..) with a new program name the program is loaded and REPLACES the existing code. Data is re-initialized and some of the process state in the kernel. Open filedescriptors are still open to the new program. Command line arguments are copied onto the newly initialized stack where main(...) will find them. Same for environment variables.

Death of a process

Parent process waits for child:

pid = wait(&status);

A parent process should always wait for children to terminate. If parents do not wait the dead child program becomes a ZOMBIE, waiting to be collected by the parent. If the parent terminates without waiting for a child the child becomes an orphan and the init process (a system process) becomes the new parent.

Literature: Alan Dix, Unix System Programming I and II, short course notes.

Process Creation and IPC

The typical shell uses the following system calls to create processes and tie them into processing pipelines:

-fork : create an exacte duplicate of the memory segments of a process and associated kernel status like open filedescriptors

-exec : load a program image from disk and start it with the arguments from the command line

-pipe : creates two filedescriptors, one for reading and one for writing

-dup : Often, the descriptors in the child are duplicated onto standard input or output. The child can then exec() another program, which inherits the standard streams. The mechanism is used by the shell to connect processes through pipes instead of letting the processes use the standard input and output files.

-wait : makes the parent block until the child is done.

Find examples here: http://www.ibiblio.org/pub/Linux/docs/linux-doc-project/programmers-guide/, The Linux Programmers Guide.

A primitive shell

```
int main (argc, **argv) {
 while (1) {       // forever
  int pid; char* command; char** arguments; int status;
  type_prompt();     // display the $ or # sign as shell prompt
  read_commands(command, arguments); // get what the user typed on the keyboard (at return)
  pid = fork();  // create a new process. If pid == 0 we are in the new child process
  if (pid == 0) {   // child
      execve(command, arguments);    // loads the new program into memory and hands over the arguments
  } else {  // continue in the parent
      waitpid(-1,,&status,0);    // parent waits for child to exit
  }
```

common

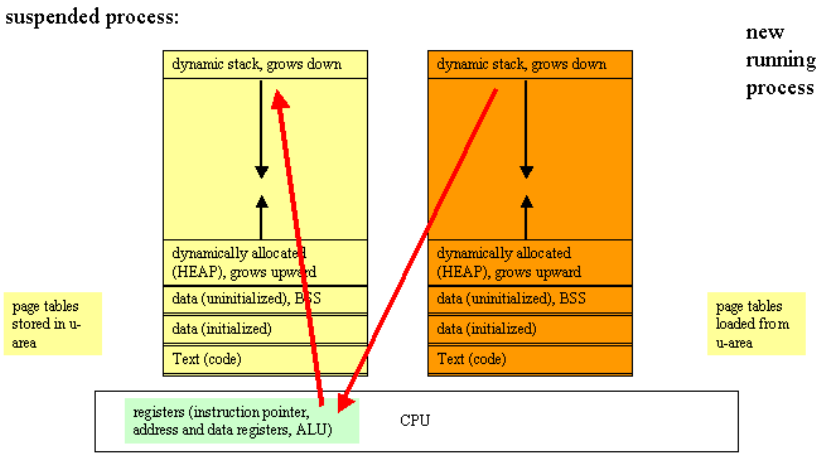
only child

only parent

This code from Tanenbaum (pg. 695) has been adjusted a little bit. It shows how simple shell architecture really is. A shell basically performs three steps again and again:

1) read users commands and arguments
2) create a new child process and make it load the requested command and execute it. The child inherits environment variables and open file descriptors
3) make the parent wait for the child to finish processing and exit.

From Process to Process: Context Switch



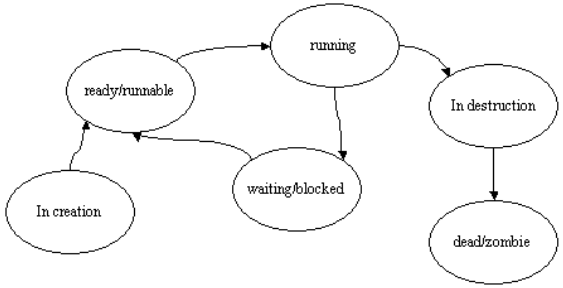
A context switch is a costly operation which involves flushing caches and storing the current state of execution within a process memory area. All information needed to continue the process after a while must be saved. To perform a context switch processing must change from user mode to kernel mode.

Reasons for context switches

1. User process gives up execution voluntarily (e.g. by using a blocking I/O system call or by calling sleep(x)
2. User process has used its time slice completely and the kernel PRE-EMPTS the process. This requires a timer interrupt into the kernel.
3. The user process runs in kernel mode (system call) and is interrupted by some high-priority device. The kernel detects that now a high-priority process is runnable and does a context switch. This requires the kernel to allow preemption in kernel mode – something most unixes do not allow.

Context switching is a technology needed for SCHEDULING processes. But it does not define WHEN to schedule a new process or WHAT process to start. These decisions are made by scheduling algorithms driven by scheduling policies.

Process States

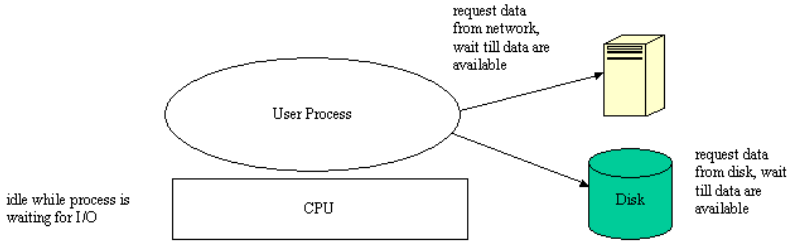


A simple state-machine diagram for possible process states. The number and kind of states are all implementation dependent but most systems know the above states. Notice that creation and destruction phases have their own state. The zombie state allows a dead process to stay in the process table e.g. because a parent process did not wait for it yet.

Process Categorizations

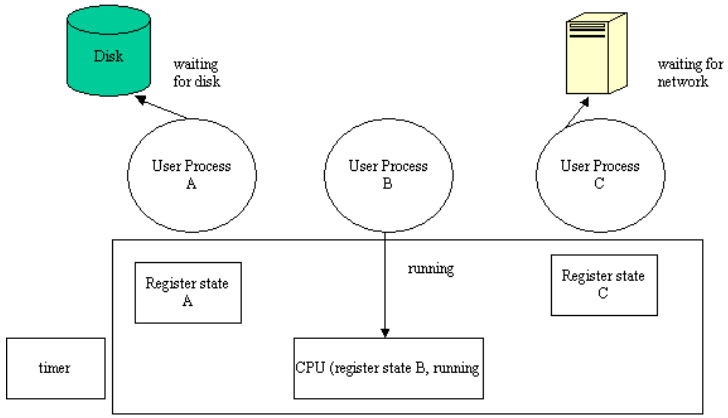
Realtime	Interactive	Batch
1. Processes are ordered by priority 2. If a process becomes runnable (e.g. a resource is now available) the kernel will check IMMEDIATELY if this process has a higher priority than the currently running process. If so, scheduling will happen immediately and the higher priority process will get the CPU.	1. Processes can have different priority and time-slices but this does NOT imply immediate reaction on events. 2. Processes may be stalled for a long time 3. The system tries to minimize reaction time for users	1. Processes can have different priority and time-slices but this does NOT imply immediate reaction on events. 2. Processes may be stalled for a long time 3. The system tries to maximize throughput and turnaround time

Why CPU (time) sharing?



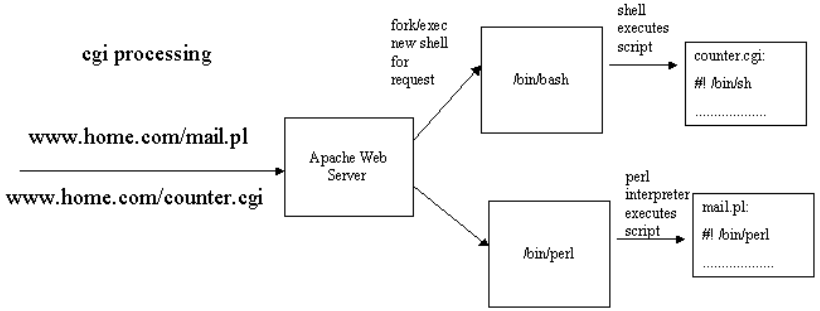
Processes frequently need external data. Those operations are so called I/O operations and take factors longer than regular instructions like add. Without an operating system the process simply idles (busy wait) for the external data to become available. During this time the CPU is blocked by the process but does not useful computations. Some processes spend more than 90% of their time waiting for resources. The idea was then to take the CPU away from a process as long as it is waiting and run some other program during this time. To allow some programs some CPU time the „TIME SLICE“ was defined as the time a process could use the CPU exclusively until it would be taken away from it. If the process had to wait for a resource it would be replaced immediately. The mechanism that does this switching of processes is called „SCHEDULING“ and can be implemented differently for real-time systems, workstations or large timesharing hosts.

Resource Management: CPU



The kernel now schedules ownership of the CPU between different processes. In this case it is clear that only process B could run because the others are blocked waiting on resources. The kernel saved their state to be able to continue the processes once the requested resources are available. A process not waiting on anything can use a whole timeslice of CPU time. A timer interrupt will tell the kernel when it is time to schedule a new process.

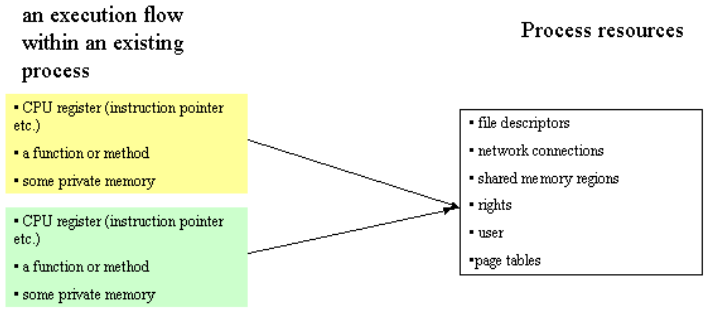
A design using several processes



CGI processing is expensive because for every request coming in a new shell or interpreter is launched which runs the requested script. Since processes are heavyweight resources most operating systems can tolerate only a certain number of large processes.

Threads

What is a Thread?

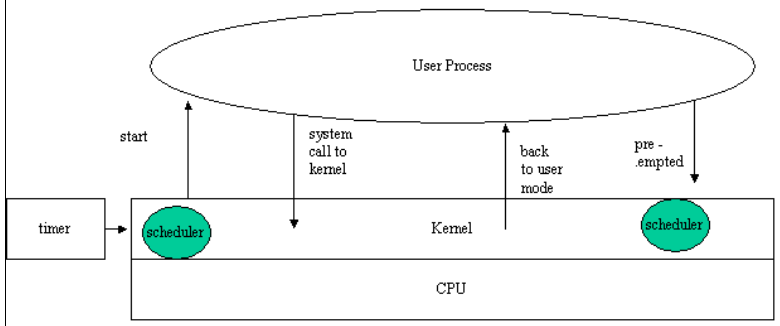


Both threads share the resources of their process. They have full access to process resources. Some threads have private storage associated which other threads cannot access.

Why Threads?

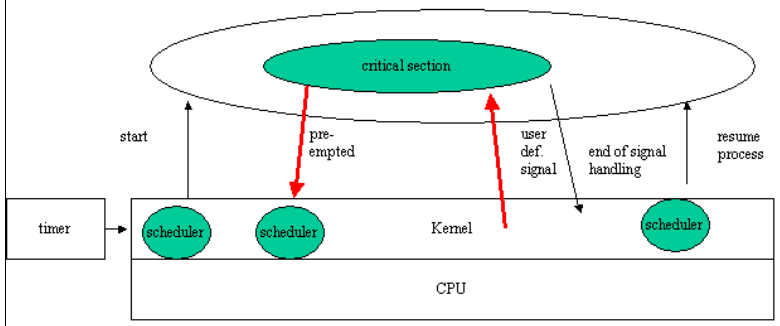
- A process contains several independent tasks which would be better represented with their own execution flow.
- A process needs more parallel execution flows. E.g. a servlet engine.
- Copying data across process borders is expensive. Threads have access to all data of the process.

When live was easy: single-threaded processes



An example of pre-emptive scheduling of user processes. At any time a user process may be scheduled if the time slice is used up, the process blocks on some resource or – with real-time systems: a process with higher priority got ready to run. The kernel will later resume the pre-empted process exactly where it left it. There is no real concurrency if only one CPU is present. But even with more CPUs the process would not run faster.

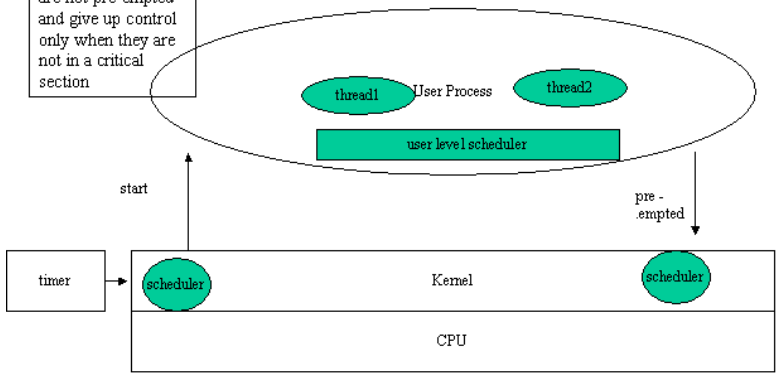
Asynchronous Signals in single-threaded processes



Unix signals are an asynchronous mechanism which can call a „handler“ function in the process. They can be caused by programming errors, user behavior or sent by other processes. While in most cases signal handling functions are called when the process is in the kernel (blocked) there is a chance that the pre-empted process was in a critical section and the signal handler now modifies values from this section. Lost updates etc. could be the result. The process can protect itself from asynchronous interruption by issuing SIGIGNORE calls. Several other race conditions can happen (see Bach, pg. 200ff)

User level threads

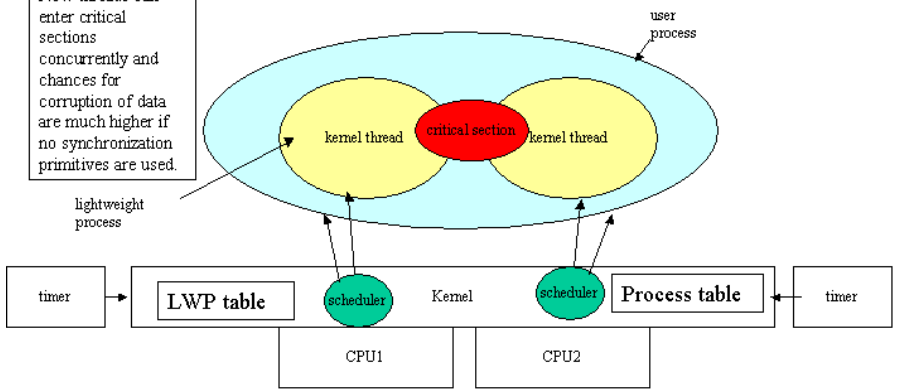
limited chance for programming errors because user threads are not pre-empted and give up control only when they are not in a critical section



With user level thread (green threads, fibres) the kernel does not schedule threads. The process itself scheduls threads which must yield control voluntarily. If a thread would do a blocking system call the whole process would get scheduled. Most user level thread libraries offer therefor non-blocking system calls. Multiple CPUs could NOT be used to speed up the process because the kernel does not know about its internal concurrency.

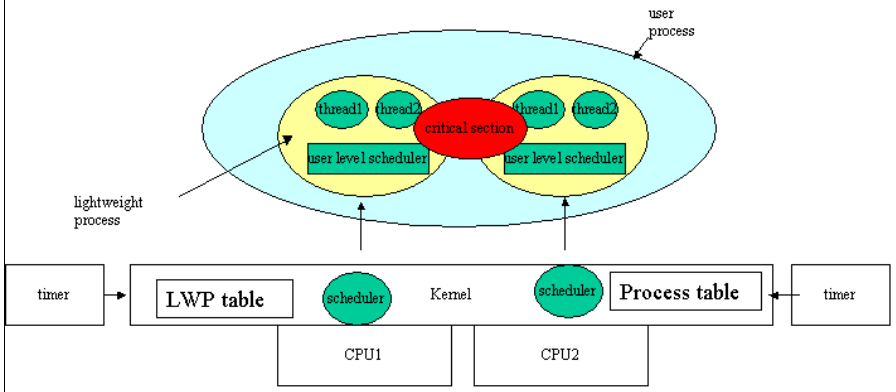
Lightweight processes (kernel threads)

Now threads can enter critical sections concurrently and chances for corruption of data are much higher if no synchronization primitives are used.



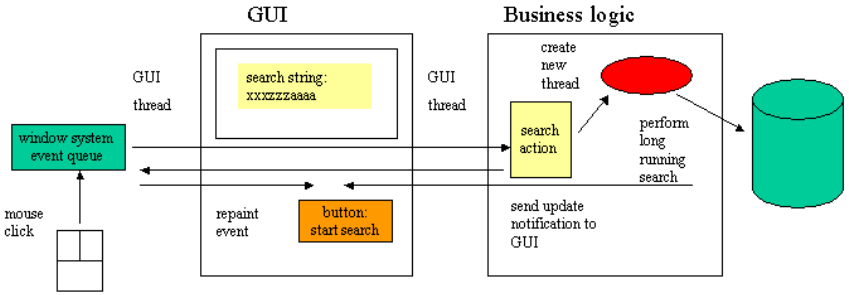
Lightweight processes are scheduled by the kernel. This means that a process can now use two CPUs REALLY concurrently by using two LWPs. This design is extremely important for virtual machines like the java VM because it can now exploit multiprocessor designs.

Kernel and User level threads



Each LWP can have regular user level threads which are scheduled under user level control. This design tries to combine the performance advantages of user level threads (extremely cheap thread context switches) with the maintenance advantages of kernel level threads which are controlled by the kernel)

A design using threads: GUI-Application



Most GUI systems use one thread to paint the GUI and process input events from keyboard or mouse. If a certain request takes a long time to process (e.g. a large database search) the GUI becomes unresponsive if the GUI thread is „abused“ to query the DB. In those cases it is better to use an extra thread for backend processing and let the GUI thread return quickly. The business logic (model) will send an update request to the GUI once the data are available. Please notice that the update request runs in a non-GUI thread and usually cannot redraw the GUI directly. Instead, it leaves a marker which causes a GUI redraw driven e.g. by timer events. Most GUI systems have asynchronous notification functions that let non-GUI threads register redraws. The reason is usually that GUI threads keep special data in threadlocal space which are not available in other threads.

Scheduling Processes or Threads

- Throughput vs. response times
- Scheduling algorithms

Throughput vs. Responsiveness

Context switching frequency is a typical example of the trade-offs made in operating system design. Each switch has an associated high overhead so avoiding too many switches is necessary to achieve high throughput.

But letting compute bound processes run for long time slices punishes I/O bound processes and makes the system feel very sluggish for the users. Good time slices are between 20 and 50 msecs with an average context switch overhead of 1msec (Tanenbaum pg. 135)

Scheduling Algorithms: Interactive Systems

- Round robin: take one process after the other
- Dynamic recalculation of priorities after CPU time used
- Based on estimates of processing time left etc.
- Lotterie (ticket based)
- Faire share
- Different Queues

Tanenbaum (pg. 147ff.) lists a large number of scheduling algorithms for interactive systems. Today most operating systems use different queues for different process types and recalculate the priority (or the number of tickets in a lottery based system) dynamically depending on how much CPU time the process already used. The more CPU time used the lower the new priority. Should two users experience the same CPU time given to their processes even if one user uses 10 processes and the other only one?

Scheduling Algorithms: realtime

- non-preemptive
- priority based
- earliest deadline first

Non-preemptive means that the processes will voluntarily yield the CPU if they cannot progress. This works because in a realtime system all processes are expected to cooperate – something not at all guaranteed for a regular multi-user system. Priority based scheduling sounds easy but can lead to strange runtime effects due to priority inversion e.g. in waiting on queued packages. Earliest deadline first is an algorithm that achieves maximum CPU use without missing deadlines as long the the overall process load is schedulable at all.

Using Priorities

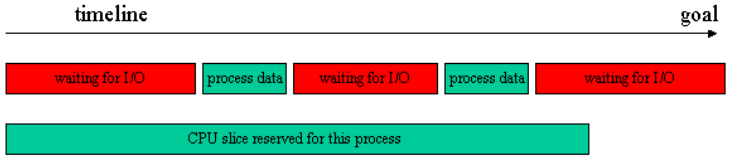
Giving processes different fixed priorities sound easy. In practice it is extremely hard to calculate the effects of different priorities in a realtime system. Two sound strategies come to mind:

a) if you absolutely want to use priorities, use a simulation software like statemate/rhapsody from www.ilogics.com to completely simulate your system.

b) if you can afford a bit more hardware, go for a bigger system that will allow you to use regular timesharing or interactive algorithms for your soft-realtime system. It will make software development much easier to treat all processes with the same priority.

And last but not least: Computer science students like to play around with new scheduling algorithms. The wins are usually in microseconds with spectacular breakdowns at certain unexpected system loads. Real professionals use a reliable and unspectacular scheduling algorithm and put all their time behind optimizing the systems I/O subsystem – this is where time is lost or won.

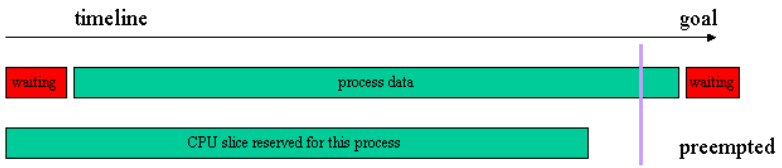
Process Behaviors (1): I/O bound



An I/O bound process spends most of its time waiting for data to come or go. To keep overall runtime short it is important to start I/O requests as soon as possible to minimize wait time. Such a process can use only minimal CPU time and still run for hours. The process metaphor can use task inherent parallelism by starting several processes doing one task. Internally it enforces strict serialized processing. This is one of the reasons why threads where invented.

The diagram shows that I/O bound processes rarely use their full share of CPU time slice.

Process Behaviors (2): compute bound

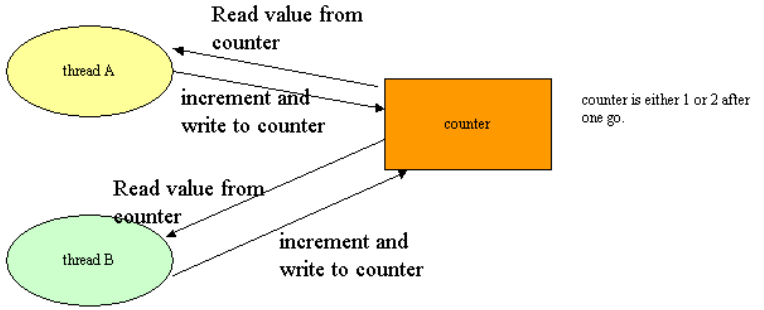


A CPU bound process does little I/O. It is not blocked waiting for resources and can therefore use its CPU slice almost fully. While being very effective with respect to throughput it causes large wait-times for I/O bound processes. Most scheduling policies therefore punish CPU bound processes by decreasing their priority or time slice to make sure that I/O bound processes can run frequently. This is a clear indicator that your scheduling strategies will depend on the typical workload of your machine and that there is no correct policy in every case.

Synchronization Issues

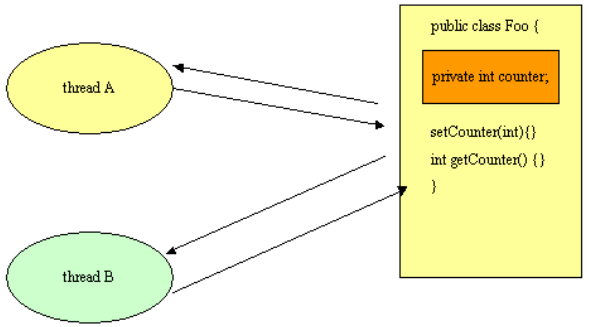
- Race Conditions
- Objects and threads
- thread-safe programs
- mutual exclusion
- test and set
- monitors

Race Conditions



Assume a thread gets pre-empted after reading the counter value. In the meantime the other thread also reads the counter and increments it. Now the first thread becomes active again and also writes the (old) incremented value into counter. The second threads increment is lost (lost update). The effect depends on when the threads are scheduled and is therefore unpredictable. Those bugs are called race-conditions and they are very hard to detect.

Objects and Threads



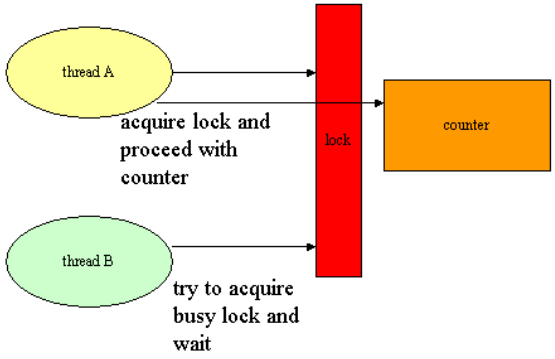
Yes, object encapsulate data. But this is not relevant for multi-threading. With respect to several concurrently operating threads the field member „counter“ from above is a GLOBAL variable if both threads share a reference to the same object. Do not confuse implementation hiding with multithreading.

Thread-safe programs

There are 3 ways to achieve thread-safe programs:

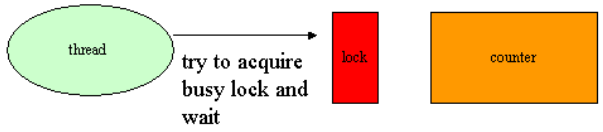
1. functions or methods do only access variables from the stack. Those are per thread only. Code that works only through stack vars is called „re-entrant“ because any number of threads can run the same code at the same time.
2. Each thread has its own set of objects and is guaranteed to be the only user of those objects. Or it uses only „threadlocal“ storage.
3. Code that does access global variables (class members, static fields, global vars in C etc.) uses protective wrappers around those to ensure exclusive access. This can be locks, semaphores or monitors.

Mutual Exclusion



One way to achieve consistency is to serialize access to resources through locks. A lock can be taken only once. The process which gets it first automatically excludes every other thread from passing the lock. Only when the lock owner finishes processing the resource and returns the lock can the other threads continue (by trying to get the lock).

Busy Waiting




```
while (true) {
  synchronized(this) {
        boolean ret = getLock(„Counter“);
        if (ret == true) break;
  }
}
```

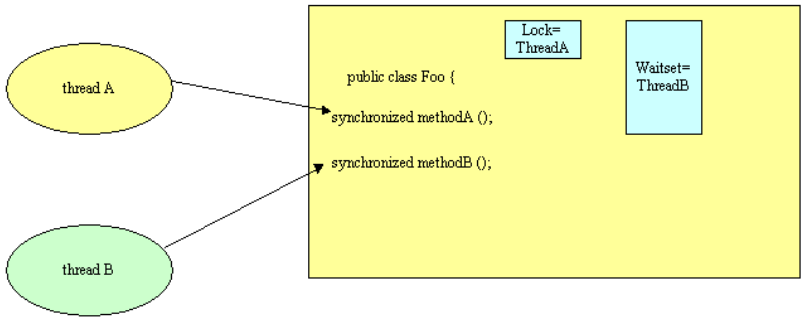
The slide shows a client doing a busy wait for resource access. This is not only a waste of CPU time but can lead to deadlocks when a system uses different fixed priorities for processes. If the lock is owned by a low priority process the high-priority busy waiter will not let the owner process run and therefore the lock will never be released (priority inversion).

Monitors

```
public class Buffer {
        private int value=0;
        private volatile boolean full=false;
        public synchronized void put (int a) throws InterruptedException { // use synchronized like this
                while(full)   // ALWAYS bracket wait with a check loop
                  wait();
                value = a; full = true;
                notifyAll(); // wake up ALL threads if they are equal
        }
        public synchronized int get() throws InterruptedException {
                int result;
                while (!full)
                  wait();
                result = false; full = false;
                notifyAll();
                return result;
        }
}
```

from Bacon/Harris. Notice that one could lock more granular using synchronized (object) and distinguish putter and getter threads. Used like above all threads will wake up, many of them just to be blocked again (e.g. all putters if the buffer is full)

Monitor Implementation



An object implementation e.g. by the java virtual machine provides a lock flag which the first thread which accesses a synchronized method or block will get. The next thread is put into the waitset of threads waiting for the lock to become available. Virtual machines try all kinds of optimizations to make monitors fast, e.g. by checking whether a thread already owns a lock or by doing atomic locks in user space instead of using kernel system calls.

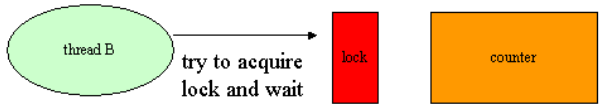
Spin Lock Class Implementation

```
class SpinLock {
private volatile boolean busy = false;
synchronized void release() { busy = false };
void acquire( ) throws InterruptedException {
  for(;;) {
    if (!busy)
      synchronized (this) {
        if (!busy) {
          busy = true;
          return; }
      }
      Thread.yield();
   }
}
```

Notice VOLATILE keyword: it tells the compiler not to optimized repeated access of the busy variable away. Also: why is synchronized used in release but only internally in acquire? This avoids a deadlock and still enforces a memory synchronization. And last but not least: Thread.yield() is NOT guaranteed to cause a thread scheduling to happen.

Example slightly modified after Doug Lea, Concurrent Java Processing. See his excellent helper classes for synchronization (resources)

Sleep/Wakeup (Guarded wait)




```
synchronized ...
while (counter <= 0) {
   wait();
}
```

The lock owner calls:

```
synchronized ..
counter++;
notifyAll();
```

The slide shows a client being put to sleep because the lock is not available. In this case it is essential that there is no chance for a race condition between lock owner and lock requester. Therefore both methods are synchronized. The lock owner MUST call wakeUpSleepers when it is done but the lock requester MUST NOT be put to sleep at exactly this moment. Otherwise it will sleep forever because it missed the wakeup call.

Why guarded waits?

```
synchronized....
while (counter <= 0) // lock to object (re-) installed
   wait();  // lock to object released
access resource...
```

There is always a possibility that several threads have been waiting on an object. A notify call will wake up one thread waiting – but there is no guarantee that it is really one of those threads that holds a resource the other threads are waiting for. Sometimes the system can also generate SPURIOUS notifications which wakes threads without the conditions being right. Of course, only one thread will get the lock to the object and then access the resource. But the other threads one after the other would access the resource later as well – without owning it.

Why notifyAll?

```
synchronized ..
counter++;
notifyAll();
```

Again, if you wake up only one thread it might be the wrong one and deadlock results. Also, priority inversion might happen if only one thread is notified to continue.

Some nasty thread problems:

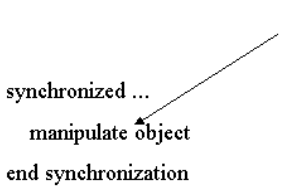
- thread interruption
- killing threads
- nested monitor problem
- serialization
- deadlocks
- atomic hardware operations

Thread Interruption

```
try {
   waiting++;
   wait();
} catch (InterruptedException e) {
  throw e; }
finally { waiting--;  notifyAll();}
```

Sometimes several threads are working together. If one gets interrupted it needs to release a semaphore variable (here: waiting) and send a notify message to wake up the other threads. They would wait forever without that notification. That's why Java enforces the InterruptedException as a checked exception.

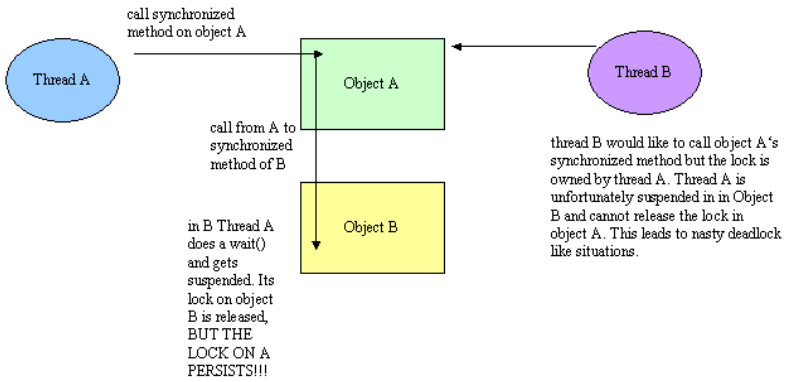
Killing a Thread



here the thread gets killed. To avoid a resource leak the java virtual machine releases the lock and kills the thread. But the object may be in an inconsistent state now because the thread did not finish.

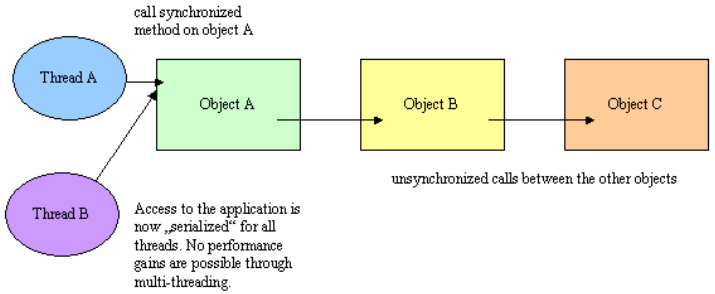
Remember: killing threads leads to an inconsistent system. Therefore the method has been deprecated.

Nested Monitor Problem



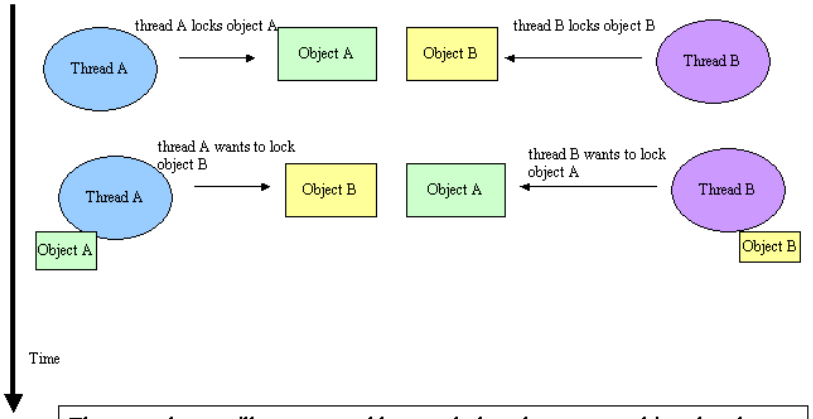
Watch for chains of synchronized methods across objects.

Serialization



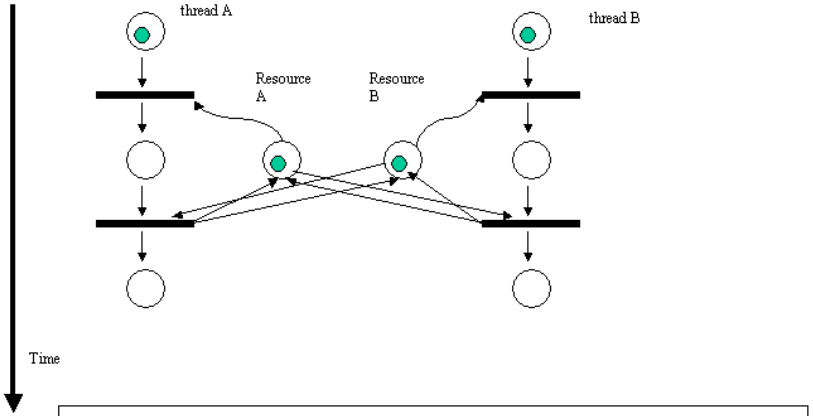
Threads only make sense if they can run in parallel – or quasi parallel. Too many synchronize statements will simply lead to complete serialization of a program: It runs just like a single-threaded application. Always grep for „synchronize" statements when you do a performance analysis. New programmers usually start using no synchronization. Then – after some mean bugs – they tend to put synchronization everywhere and slow things down until they learn how to use it properly.

Deadlock



The second step will create a problem: each threads wants an object that the other one already owns. No thread can progress and everything stalls. Think about solutions to this problem. Will your solution be stable? I.E. work under different conditions and after code changes as well? We will make some exercises on this.

Deadlock Description with Petri Nets



A petri net takes a transition if all input phases have tokens. In this case both resource A and B would lose their token. The next transition is not possible because the other resource token is not available. This deadlock depends on the order of transitions which is non-deterministic – a property well expressed by petri nets. (see resources: Uwe Schoenig, Ideen der Informatik pg. 62 from where this example was taken)

Atomic Test and Set Instructions

```
label: tset r0, memoryAddress
       cmp r0, 1
       jeq label
```

tset is a so-called atomic instruction. It takes the content of memoryAddress and puts it in register r0. If register r0 has the value 0 it will put a ONE at the location memoryAddress. If r0 was already one it leaves it as it is. The magic lies in the fact that the hardware guarantees that swapping 0 and 1 ad memoryAddress IS AN INDIVISIBLE or ATOMIC OPERATION. No other process can interfere here (e.g. read or write to memoryAddress during this time). Multiprocessor systems rely on those tset statements but they are usefully on any system. Please not that implemented like this the code from above would do a busy wait on the lock. But this logic can be coupled with a context switch and notification logic which gives us the concept of semaphores.

A word of caution

Most synchronization mechanisms rely on the fact that all clients go through those mechanisms to get ownership of a lock. In many cases it would be possible for clients to simply try to access the resource directly, bypassing the mutual exclusion mechanism. If this happens data inconsistencies are certain. If you experience strange inconsistencies and your code looks absolutely proper: search for clients which go through

a) no locks at all to access the resource (shoot the developer)

b) go through a private locking mechanisms to access the resource (shoot the developer but torture him first)

This is based on the experience of searching 2 month for a bug in a multi-processing system with real-time slave controllers. The communication between host and slaves became inconsistent every once in a while. The reason was that a tty device driver used a private lock mechanism to lock the system bus. If somebody used a terminal attached to my test systems my dual-ported communication channel became inconsistent....

Needless to say that it takes an endless amount of debugging and testing until you start questioning totally unrelated code...

Theory: Concurrency Models

- Shared State Model (what you have seen on the previous slides. Some people claim that this model is error prone. It works only on local machines.)
- Message Passing Concurrency: basically the approach from distributed computing ported back to local machines. It avoids sharing state completely. Objects communicate via sending messages to ports.
- Actor Model: Similiar to message passing concurrency. The symbian OS uses the concept of active objects to avoid multithreading but it is non-pre-emptive and requires design changes.

Many modern or research languages and operating systems use different concurrency models. Take a look at the E-language or OZ (see resources) for newer concepts (promises etc.). And take a look at the phantastic book by Peter van Horn and Seif Haridi and the excellent web page which accompanies it)

Resources (1)

- Doug Lea, Concurrent Programming in Java, Second Edition. Uses design patterns. A threadpool implementation is available, together with other synchronization primitives in Java. Do not mess with threads without this book. http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html . Please look at his excellent helper classes for all kinds of synchronization problems and a high performance threadpool implementation.
- Raphael A. Finkel, An Operating System Vademecum, find it online at: ftp://ftp.cs.uky.edu/cs/manuscripts/vade.mecum.2.pdf
- A good class on Operating Systems: http://www.doc.ic.ac.uk/~wjk/OperatingSystemsConcepts/ with simple kernel tutorial (bochs)
- Alan Dix, Unix System Programming I and II, short course notes.

Resources (2)

- Concurrency: State Models & Java Programs, Jeff Magee and Jeff Kramer, http://www-dse.doc.ic.ac.uk/concurrency/ with nice applets and code for various concurrency examples
- Peter Van Roy and Seif Haridi. Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge, MA, 2004. All sorts of programming concepts using the OZ language. This book will make the difference if you really want to understand computer science. Fundamental like „Structure and Interpretation of computer programs“. There is a draft available from wayback engine. (almost 1000 pages). Excellent additional material (courses etc.) at: http://www.info.ucl.ac.be/people/PVR/ds/mitbook.html
- Active Objects in Symbian OS: http://www.symbian.com/developer/techlib/papers/tp_active_objects/active.htm . Gives an overview of multitasking concepts as well.
- Gerlicher/Rupp, Symbian OS. Very good introduction to Symbian OS (with active objects, communication etc.) GERMAN.