Process

- a program "alive" (in the execution)
- a processor abstraction + resources provided by the OS to a user program
- OS process management:
  - supports user creation of processes and interprocess communication (IPC)
  - allocates resources to processes according to specific policies (process scheduling)
  - interleaves the execution of multiple processes to maximize CPU utilization (process switching)

Process state diagram

- ready (in memory)
- running
- suspended (swapped out)
- blocked
- activated
- dispatched
- timeout
- event occurred
- wait for event

Process image

- the physical representation of a process in the OS
- an address space consisting of code, data and stack segments
- a process control structure
  - identification: process, parent process, user
  - context of the execution: PC, SP, other registers
  - control: scheduling (state, priority), resources (memory, opened files), IPC
Execution mode

- most processors support at least two modes of execution: privileged (kernel-mode) and non-privileged (user-mode) for protection reasons
- the portion of the OS executed in kernel-mode is called kernel (basic functions which must be protected from interference by user programs)
- in kernel-mode the OS can do anything (no protection)
- user processes execute in user-mode
- in user-mode a process can access only its address space

Interrupts and traps

- Interrupts: asynchronous
  - external events (not related to the processor state) which occur independently of the instruction execution in the processor
  - can be masked (specifically or not)
- Traps: synchronous
  - conditionally or unconditionally caused by the execution of the current instruction
- Interrupts and traps force the processor to save the current state of execution and jump to a new state indicated in the interrupt vector

Mode switching

- one or several bits in the processor status word (PS) indicate the current (sometimes also the previous) mode of execution
- these bits can be changed explicitly while in kernel-mode but not in user-mode
- the processor status word can be loaded from the interrupt vector along with the address of the interrupt handler
- by setting the kernel mode bit in the interrupt vectors user processes enter kernel-mode to execute interrupt handlers

Signals

- UNIX mechanism to upcall a handler indicated by the user process when an event occurs: segmentation violation, message arrival etc...
- the kernel handles the event first, then puts an activation record on the user-level stack corresponding to the event handler
- when the user process is scheduled next it executes the handler first
- from the handler the user process returns to where it was when the event occurred
System calls

- to request an OS service a process issues a system call
- in monolithic OS a system call is implemented as a trap
- when executing the user program code a process runs in user-mode
- when executing the OS code corresponding to a system call a process runs in kernel-mode
- in microkernel OS a system call is implemented as an IPC

System call in monolithic OS

- code for read system call
- kernel mode
- interrupt vector for trap instruction
- in-kernel file system (monolithic OS)
- user mode

Process creation

- can be performed by the OS transparently to the user or can be made visible to the user as a system call
- in UNIX a process (parent process) can spawn another process (child process) by calling fork
- the child process is created as a copy of the parent process (process image and process control structure) except for the identification and scheduling state
- parent and child processes run on two different address spaces => by default no memory sharing

Efficient process spawning

- copy the entire process image is time consuming, optimizations are possible
- code segment can be made read-only and shared
- copy-on-write for data segments
  - at fork time the data segment is mapped read-only and shared between the child and the parent process
  - the copy is performed lazily, in page increments at first write
Process switching

- processor can switch modes from user to kernel and back to user while running the same process
- process switching is the act of taking a process off the processor and replacing it with another one that is waiting to run
- process switching occurs in kernel-mode and is a result of process scheduling that is performed when one of the following occurs:
  - time quota allocated to the process expires
  - a blocking system call
  - memory fault due to a page miss

Process switching in UNIX

- each process has a kernel stack which is accessible only to the kernel but is part of the process image
- each process has a page table which maps both its memory image and the kernel memory image
- when switching mode from user to kernel, process switches to its kernel stack but doesn’t change the page table

Process switching in UNIX (cont’d)

- the execution state of the old process is saved on its kernel stack
- the kernel switches from the old address space to the address space of the process scheduled to run next
- the execution state of the new process to run is restored from its kernel stack

Inter-process communication

- most OSs provide several abstractions for inter-process communication: message passing, shared memory, etc.
- communication requires synchronization between processes (i.e. data must be produced before it is consumed)
- synchronization can be implicit (message passing) or must be explicit (shared memory)
- explicit synchronization can be provided by the OS (semaphores, monitors, etc.) or can be achieved exclusively in user-mode (if processes share memory)
Message passing

- protected communication between different address spaces
  
  send (destination, message)
  
  receive (source, message)
  
  synchronization on send:
  
  - asynchronous (non-blocking send)
  
  - blocking send until the message “is sent”
  
  - blocking send until the message is received
  
  synchronization on receive:
  
  - asynchronous (non-blocking receive)
  
  - blocking receive until message arrives

Message passing (cont’d)

- message addressing
  
  - direct: specify the process
  
  - indirect: specify the queue
  
  - mutual exclusion using messages
  
  - non-blocking send
  
  - blocking receive

Message passing implementation

- two copy operations in a conventional implementation

Shared memory implementation

- no copying but synchronization is necessary
Threads

- the process concept incorporates two abstractions:
  - a virtual processor (an execution context) and a resource ownership (an address space, opened files etc.)
- within an address space we can have more units of execution: threads
- all the threads of a process share the same address space and the same resources

Multithreaded environment

- Process
  - a virtual address space that holds the process image
  - all resources allocated: IPC channels, files etc.
- Threads
  - an execution state: running, ready, etc.
  - an execution context: PC, SP, other registers
  - a per-thread stack

Threads vs. Processes

- advantages
  - operations on threads (creation, termination, scheduling, etc.) are cheaper than the corresponding operations on processes
  - inter-thread communication is supported through shared memory without kernel intervention
- disadvantages
  - easy to introduce race conditions
  - synchronization is necessary

Posix thread (Pthread) API

- thread creation and termination
  - `pthread_create(&tid,NULL,start_fn,arg);`
  - `pthread_exit(status)`
- thread join
  - `pthread_join(tid, &status);`
- mutual exclusion
  - `pthread_mutex_lock(&lock);`
  - `pthread_mutex_unlock(&lock);`
- condition variable
  - `pthread_cond_wait(&c,&lock);`
  - `pthread_cond_signal(&c);`
Condition variables (example)

thread 1

```c
pthread_mutex_lock(&lock);
while (!my-condition);
    pthread_cond_wait(&c,&lock);
do_critical_section();
pthread_mutex_unlock(&lock);
```

thread 2

```c
pthread_mutex_lock(&lock);
my-condition = true;
pthread_mutex_unlock(&lock);
```

More on synchronization

- critical section: a section of code which reads/writes shared data
- race condition: potential for interleaved execution of a critical section by multiple threads => results are non-deterministic
- mutual exclusion: synchronization mechanism to avoid race conditions by ensuring exclusive execution of critical sections
- deadlock: permanent blocking of threads
- starvation: execution but no progress

Requirements for ME

- no assumptions on hardware: speed, # of processors
- execution of CS takes a finite time
- a thread/process not in CS cannot prevent other threads/processes to enter the CS
- entering CS cannot be delayed indefinitely: no deadlock or starvation

Conventional solutions for ME

- software reservation: a thread must register its intent to enter CS and then wait until no other thread has registered a similar intention before proceeding
- spin-locks using memory-interlocked instructions: require special hardware to ensure that a given location can be read, modified and written without interruption (i.e. TST: test&set instruction)
- OS-based mechanisms for ME: semaphores, monitors
Software reservation

works both for unprocessors and multiprocessors but have overheads and memory requirements

multiple algorithms: Dekker, Peterson, Lamport \((2\,ld + 5\,st)\)

start:
  \(b[i] = \text{false};\)
  for \(j = 1\) to \(N\) await(b[j]==false);
  if (y \(!=\) i) {
    \(b[i] = \text{false};\)
    await (y==0);
    goto start;
  }

y = i;
if (x \(!=\) i) { /* collision */
  b[i] = \text{false};
  for \(j=1\) to \(N\) await(b[j]==false);
  if (y \(!=\) i) {
    await (y==0);
    goto start;
  }
}

CRITICAL SECTION

y = 0;
if (x \(!=\) i) { /* collision */
  b[i] = \text{false};
}

Spin-locks (busy waiting)

inefficient on unprocessors: waste CPU cycles

on multiprocessors cache coherence effects can make them inefficient

Problem:

\[
\begin{align*}
\text{lock} &= \text{false} \quad /\!\!/ \text{ init } /\!
\text{while (TST(lock)==TRUE)}; \quad /\!\!/ \text{ busy waiting to get the lock}
\text{cause bus contention} /\!
\text{lock} &= \text{false}; \quad /\!\!/ \text{ unlock } /\!
\end{align*}
\]

1st solution:

\[
\begin{align*}
\text{lock} &= \text{false} \quad /\!\!/ \text{ init } /\!
\text{while (lock == TRUE || TST(lock)==TRUE);} \quad /\!\!/ \text{ spinning is done in cache if lock is busy } /\!
\text{lock} &= \text{false}; \quad /\!\!/ \text{ unlock } /\!
\end{align*}
\]

Cache coherence effect

TST causes cache invalidations even if unsuccessful

1st solution: keeps spinning in the cache as long as the lock is busy

at release, lock is invalidated, each processor incurs a read miss

first processor resolving the miss acquires the lock

those processors which pass the spinning in the cache but fail on TST generate more cache misses

partial solution: introduce random delays

Spin-locks with delays

2nd solution:

\[
\begin{align*}
\text{lock} &= \text{FALSE} \quad /\!\!/ \text{ init } /\!
\text{while (lock==TRUE || TST(lock)==TRUE);} \quad \{
\text{while (lock==TRUE)}
\quad \text{delay();}
\}
\text{lock} &= \text{FALSE}; \quad /\!\!/ \text{ unlock } /\!
\end{align*}
\]

if some processor acquires the lock during the delay then the processor can resume spinning rather than executing an unsuccessful TST
Spinning vs blocking

- spinning is good when no other thread waits for the processor or the lock is quickly released
- blocking is expensive but necessary to allow concurrent threads to run (especially if one happens to hold the lock)
- combine spinning with blocking: when a thread fails to acquire a lock it spins for some time then blocks
- if the time spend in spinning is equal to a context switch the scheme is 2-competitive
- more sophisticated adaptive schemes based on the observed lock-waiting time

Kernel emulation of atomic operation on uniprocessors

- kernel can emulate a read-modify-write instruction in the process address space because it can avoid rescheduling
- the solution is pessimistic and expensive
- optimistic approach:
  - define restartable atomic sequences (RAS)
  - practically no overhead if no interrupts
  - recognize when an interrupt occurs and restart the sequence
  - needs kernels support to register (RAS) and detect thread switching in RAS

TST emulation using RAS

```c
Test_and_set(p) {
    int result;
    result = 1;
    BEGIN RAS
        if (p==1)
            result = 0;
        else
            p = 1;
    END RAS
    return result;
}
```

Thread implementation

- user-level threads
  - implemented as a thread library which contains the code for thread creation, termination, scheduling and switching
  - kernel sees one process and it is unaware of its thread activity
  - can be preemptive or not (coroutines)
- kernel-level threads (lightweight processes)
  - thread management done by the kernel
User-level vs. kernel-level threads

- **Advantages of user-level threads**
  - Performance: low-cost thread operations (do not require crossing protection domains)
  - Flexibility: scheduling can be application specific
  - Portability: user-level thread library easy to port

- **Disadvantages of user-level threads**
  - If a user-level thread is blocked in the kernel, the entire process (all threads of that process) are blocked
  - Cannot take advantage of multiprocessing (the kernel assigns one process to only one processor)

User-level vs. kernel-level threads (cont’d)

User-level thread implementation

<table>
<thead>
<tr>
<th>Operation</th>
<th>User-level threads (usec)</th>
<th>Kernel-level threads (usec)</th>
<th>Processes (usec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>null fork</td>
<td>34</td>
<td>948</td>
<td>11,300</td>
</tr>
<tr>
<td>signal-wait</td>
<td>37</td>
<td>441</td>
<td>1,840</td>
</tr>
</tbody>
</table>

Thread/process operation latencies
Kernel support for user-level threads

- user-level threads can execute within the context of traditional processes or kernel-level threads which are viewed as virtual processors
- both are wrong abstractions in supporting user-level threads
- mismatch between where the scheduling information is available (user) and where scheduling on real processors is performed (kernel)
- when the virtual processor is blocked the corresponding physical processor is lost to all user-level threads although there may be some ready to run and idle processor(s)

Scheduler activations

- kernel allocates processors to user address spaces as scheduler activations
- each address space contains a user-level thread system which controls the scheduling on the allocated processors
- kernel notifies an address space whenever the number of allocated processors changes or when a scheduler activation is blocked due to the user-level thread running on it
- the address space notifies the kernel when it needs more or fewer scheduler activations (processors)
Scheduler activations vs kernel-level threads

- with kernel-threads a blocked user-level thread is resumed by the kernel along with the kernel-level thread
- with scheduler activations, the user-level scheduler removes the user-level thread from the scheduler activation upon notification
- a new scheduler activation (corresponding to the allocated processor) is sent with the notification, in which the user-level scheduler can schedule another user-level thread
- invariant: at any time there are as many running scheduler activations as processors allocated to the address space