Operating Systems
2010/2011

Kernel Activities

Johan Lukkien
Agenda

• What is required to implement a kernel?

• General system calls

• Scheduling
Provided service (system call API) to applications (running processes)

OS Kernel

Required service from platform (processor, other hardware)
Provided service (system call API) to applications (running processes)

OS Kernel

Required service from platform (processor, other hardware)

Process management
- `fork()`, `exec()`, `exit()`, `wait()`

Thread management
- `pthread_create()`, `pthread_join()`, ...

Synchronization
- semaphores, condition variables, ...

Communication
- `pipes`, `sockets`, ....
Provided service (system call API) to applications (running processes)

OS Kernel

Required service from platform (processor, other hardware)

- Kernel/user mode
- Trap & interrupt mechanism
- Memory management, protection
- Fast exclusion mechanism
Assumptions on platform (required service)

- Processor supports two modes:
  - supervisor (kernel) and user mode
    - user mode does not allow access to critical resources like device registers, memory management registers etc.

- Regular trap, exception, interrupt mechanisms
  - switch to kernel mode, save status (= return address, stack), load handler address and kernel stack, execute handler, return
    - trap: internally-caused (under program control) switch to the kernel
    - exception: error-induced switch to the kernel
      - illegal reference, illegal instruction, ..... 
    - (regular) interrupt: external-event caused switch to the kernel
How does the kernel get control?

Asynchronous interrupt: breaking control flow of application

Synchronous interrupt: as part of control flow of application

22-Nov-10
Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking
Assumptions on platform *(required service)*

- Processor supports two modes:
  - supervisor (kernel) and user mode
    - user mode does not allow access to critical resources like device registers, memory management registers etc.

- Regular trap, exception, interrupt mechanisms
  - switch to kernel mode, save status (return address, stack), load handler address and kernel stack, execute handler, return
    - trap: internally caused (under program control) switch to the kernel
    - exception: error induced switch to the kernel
      - illegal reference, illegal instruction, ..... 
    - (regular) interrupt: external-event caused switch to the kernel

- Support of memory protection, typically via a *memory management unit*
  - the MMU determines the current view on the memory: the *context*
    - the pages that may rightfully be addressed in user mode
    - determined by the setting of memory management registers
  - user mode references outside designated area result in an exception
Mode and context

• A mode switch
  – is needed for any event that requires kernel activity
  – changes the processor mode and loads a new program counter
    • mostly changing stack point as well
  – must usually be followed by saving some additional processor state
  – is a fast operation (executed in hardware)

• A context switch
  – changes the *memory management* settings
  – changes which user code is executing
  – is needed, besides one or more mode switches, *for only some events*, e.g. if a process switch is needed
  – is a slow operation

• Hence, we would like to restrict the majority of the event handling to just a mode switch, plus corresponding handler
Assumptions on platform (*required* service)

- Processor supports two modes:
  - *supervisor (kernel) and user mode*
    - user mode does not allow access to critical resources like device registers, memory management registers etc.

- Regular trap, exception, interrupt mechanisms
  - *switch to kernel mode, save status (return address, stack), load handler address and kernel stack, execute handler, return*
    - trap: internally caused (under program control) switch to the kernel
    - exception: error induced switch to the kernel
      - illegal reference, illegal instruction, ....
    - (regular) interrupt: external-event caused switch to the kernel

- *Support of memory protection, typically via a memory management unit*
  - determines the current view on the memory: the *context*
    - the pages that may rightfully be addressed in user mode
    - determined by the setting of memory management registers
  - user mode references outside designated area result in an exception
Assumptions on platform (required service)

- Processor supports two modes:
  - supervisor (kernel) and user mode
    - user mode does not allow access to critical resources like device registers, memory management registers etc.

- Regular trap, exception, interrupt mechanisms
  - switch to kernel mode, save status (return address, stack), load handler address and kernel stack, execute handler, return
    - trap: internally caused (under program control) switch to the kernel
    - exception: error induced switch to the kernel
      - illegal reference, illegal instruction, ....
    - (regular) interrupt: external-event caused switch to the kernel

- Support of memory protection, typically via a memory management unit
  - determines the current view on the memory: the context
    - the pages of virtual memory that may rightfully be addressed in user mode
    - determined by the setting of memory management registers
  - user mode references outside designated area result in an exception

- Binary semaphore basics: basic exclusion mechanisms, simple implementations:
  - ability to selectively inhibit / disable interruption
  - spinlocks
Binary semaphores

- A binary semaphore assumes only the values 0 and 1
  - \( P \) and \( V \) must strictly alternate, calling \( V \) without a preceding \( P \) is an error
  - This does not mean that the implementation uses just a binary variable

- A BS can be mapped onto hardware instructions
  - test&set instruction: \( T&S(s,x) \): \(<x := s; s := 1>\)
    - \( P(s) \): \{do \ T&S(s,x) \ while \ x = 1\}
    - \( V(s) \): \( s := 0 \)
    - \( x \) is a private variable of caller
  - fetch&add: \( F&A (s,x) \): \(<\text{return } (rtn)>\)
    - \( rtn \) is a private variable of caller
    - definition \( P(s), V(s) \): exercise
  - compare&swap:
    - \( C&S (b,o,n,s) \): \(<b := (s=o); \ if \ b \ then \ s := n \ fi>\)
    - \( o \) and \( n \) stand for: old value, new value; hence, \( C&S \) assigns a new value to \( s \) if the current value equals the old value (“\( s \) did not change since last inspection”); \( b, o \) and \( n \) are local (private) variables of caller
    - definition \( P(s), V(s) \): exercise
  - using this for exclusion gives busy waiting

- The general term for this type of busy waiting is a spinlock
Binary semaphores

• The prime use of binary semaphores is for exclusion

• Spinlocks yield busy waiting
  – only acceptable if duration is guaranteed to be short
  – useful between machines
  – useless for activities on same machine (why?)

• Exclusion between activities on same machine:
  – make sure no interruption (typically, asynchronous) is possible during critical section
  – hence, enable/disable(inhibit) asynchronous interruption (= external interrupts)
    • actually, only those that are problematic for that critical section
    • P(s): inhibit interrupts (older Linux source/x68 arch: cli)
    • V(s): enable interrupts (older Linux source/x68 arch: sti)
  – useless between machines (why?)
  – note: exceptions (traps) in these critical sections must be ruled out!
Where to use these binary semaphores?

• In a user process?
  – avoids system call overhead for e.g. a semaphore
  – busy waiting: implies (potential) waiting on another machine
    • is pointless when –incidentally- competing processes run on same machine
    • consumes processing power, blocking progress of other processes
  – interrupt inhibition: implies disabling all other activity (including kernel)
    • pointless in the presence of several machines

• Within the OS kernel?
  – as part of the implementation of system calls
    • for access to system data structures
  – needs both mechanisms (why?)
  – critical sections must be brief, and must terminate
    • in case of longer-lasting or blocking waits: use queuing of the waiting process
    • typical examples of the use of these binary semaphores include the manipulation of these queues
Agenda

• What is required to implement a kernel?

• General system calls

• Scheduling
Synchronization inside kernel

• Concurrent activities:
  – system calls
  – interrupts
  – multiple instances of these, with multiprocessors

• Use combination of interrupt inhibition and spinlocks
  – spinlock: \( N \)-party, hardware-based semaphore
    • often optimized to \( N \) parties, beyond the test&set
    – carefully decide what to protect
      • the more global the spin-locking, the more delay other processors incur
        – (could lock the entire OS at the expense of having no concurrency)
      • the longer the interrupt inhibition, the higher the probability to miss an interrupt
        – however, there is no direct performance loss
  – the order: inhibit; lock and not: lock; inhibit
    • why?
From 1st lecture: Example interaction: read data from file

- Interaction is named: system call
- Execute C statement: \( \text{status} = \text{read}(\text{fd}, \text{buffer}, \text{nbytes}) \)
  - read \( \text{nbytes} \) bytes from \( \text{fd} \), storing it in \( \text{buffer} \)
  - is part of a program running on top of the OS (e.g. banking program)

- memory: kernel space/user space
  - kernel space only accessible with processor in kernel mode
- parameters: either via registers or via memory
  - 1-3: pushing parameters
  - 4: call library function
  - 5: put code for \text{read} in reg.
  - 6: trap (Linux@x86: int 0x80): switch mode and call trap handler
  - 7: handler calls read function handler
  - 8: handler performs read actions (a.o. store data at address). Here suspension of the calling process may occur if data needs to come from an io device.
  - 9-11: control back to caller
System call structure

- Consider the typical condition synchronization
  - $P(m)$;
  - while not condition do wait $(m, c)$ od;
  - critical section;
  - signals;
  - $V(m)$

- This also models a **blocking system call**
  - $P$ and $V$ are implemented by inhibition and spinlocks
  - Waiting is short (provided the critical section is short)
  - Between the $P(m)$ and the waiting some action may occur as well – typically, just reading

- In some system calls, the activity that makes the condition *true* is started as new activity; this activity then signals
  - $P(m)$;
  - Initial access to datastructures, typically for checking;
  - while not condition do start task to make condition true; wait $(m, c)$ od;
  - system call remainder;
  - signals;
  - $V(m)$
Outline of system call handling

**Trap Handler:** (continue here after mode switch)

Check Validity of parameter;

Call SystemCalls [parameter] (arguments of call);

{ SystemCalls is a table of functions; }

Execute return path; { check on pending signals, remaining kernel tasks, the need to reschedule }

**Call:**

{ next slide }

Example: `read(fd, buffer, cnt)`
Outline of system call handling

**Call:**
- Check arguments;
- Access relevant kernel data – use interrupt inhibition, locking when required, at the right level;
- **while Blocking Condition do** { or: **if Blocking Condition then** }
  - Setup system to make condition become true;
  - Add *Self* to queue associated with resource or event corresponding to *Blocking Condition*;
  - Call Scheduler; { the current process is stopped and put in blocked queue, another one started
    the critical section entered in this call is released (last); }
  { call is restarted here; may need to resume critical section; notice: restart mode is kernel mode }
- **od;**

**Final Operations:** { e.g. store results in process }
- **if** any processes can be continued as result of state changes that this call has brought about **then**
  - Release these processes; { need to know where they are stored! }
- **fi;**
- Release critical section entered in this call;
Outline of system call handling

Call:
Check arguments;
Access relevant kernel data – use interrupt inhibition, locking when required, at the right level;
while Blocking Condition do { or: if Blocking Condition then }
  Setup system to make condition become true;
  These marked parts access process management data structures and need further associated protection (locking)
  Add Self to queue associated with resource or event corresponding to Blocking Condition;
  Call Scheduler; { the current process is stopped and put in blocked queue, another one started
 the critical section entered in this call is released (last), } { call is restarted here; may need to resume critical section; notice: restart mode is kernel mode }
  od;
Final Operations; { e.g. store results in process }
  if any processes can be continued as result of state changes that this call has brought about then
    Release these processes. { need to know where they are stored }
  fi;
Release critical section entered in this call;
Discussion

• Protection:
  – interrupt inhibition refers to the executing processor
    • after that point there is no interference with competing interrupt handlers
  – locking refers to exclusion between processors
    • after that point exclusion between processors is guaranteed as far as code sections/data
guarded by that lock are concerned – this is the level of protection

• Blocking Condition refers to the state needed to complete the call,
  – e.g. whether a disk block is available or a message has been left behind

• Similar as for condition variables, the evaluation of the condition and subsequent
  recording of the process in a queue must not lead to a race condition
  – OS-specific organization, typically through the given exclusion mechanism and
    perhaps other primitives inside the kernel

• Notice the structure of condition synchronization
  – check and block where the condition is needed
  – signal where others may be released
Discussion (cnt’d)

• A queue per blocking event increases lookup speed at the check-and-release side

• After wakeup, execution proceeds after the call to the scheduler (or within the scheduler), in kernel mode

• Critical sections are supposed to be short
  – actual waiting is in blocked mode
  – never prolonged busy waiting, only the locking

• Conditions are assumed to be unstable – otherwise an if-statement suffices

• Upon event occurrence the waiters are moved to the ready queue
  – event: another call or an interrupt routine that signals completion

22-Nov-10
Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking
Discussion (cnt’d)

• Notice the race condition (or its avoidance):
  – Add *Self* to queue; Release critical section & schedule another process

• It is to be expected that system calls may need several kernel data structures
  – hence, more locks
  – need to keep track of acquired locks *and their order*

• One way to address this repeated, and possibly nested locking is to have locks at a less detailed level
  – kernel lock – would make the entire kernel uninterruptable
  – lock per kernel data structure

• Objects that share a lock will have mutual exclusion
Unix process state transition diagram

- Note: blocking is in kernel mode
Examples

- Simple call without blocking
- Semaphore operations
- Condition variable operations
Example: return an element of the PCB

- `getuid()` on Linux (‘get userid’)

- Body of the system call: just return the value
  - `return (self->uid)`
  - or: store this value at the place in the process where this return value is expected

- No exclusion needed
- No blocking occurs
- Simple mode switch suffices
Example: implementing general P/V

\[ P(s): \]
\begin{align*}
& \text{EnterCS; \{ inhibit interrupts; and use } s -> \text{lock (spinlock) for this semaphore } \} \\
& \text{while } s -> \text{val} = 0 \text{ do} \\
& \quad \text{Store self in } s -> \text{queue;} \\
& \quad \text{Call Scheduler; \{ stop current (putting it in blocked queue!); EnterCS, start other; \} } \\
& \quad \text{EnterCS; \{ resume critical section } \} \\
& \text{od; } \\
& s -> \text{val} := s -> \text{val} - 1; \\
& \text{LeaveCS \{ Release } s -> \text{lock and enable interrupts } \} \\
\end{align*}

\[ V(s): \]
\begin{align*}
& \text{EnterCS; \{ inhibit interrupts; and use } s -> \text{lock (spinlock) for this semaphore } \} \\
& s -> \text{val} := s -> \text{val} + 1; \\
& \text{if not empty}(s -> \text{queue}) \text{ then} \\
& \quad \text{Release one from } s -> \text{queue;} \\
& \text{fi} \\
& \text{LeaveCS; \{ Release } s -> \text{lock and enable interrupt } \} \\
\end{align*}

- the semaphore is here a shared data structure (in the kernel)
  - containing lock, value and queue, in this case
  - created by API calls like the ones presented before
  - \( s \) is a descriptor, a pointer to this data structure
- unfairness possible (how?)
- \( P(s) \) has only the waiting part of the call; \( V(s) \) has only the releasing part
- manipulating the process records in the Scheduler and the Release needs protection
- optimized version
- released processes have precedence over newcomers (argument?)
Implementation of condition variables

• Need to implement all operations
  – \texttt{wait}(m, c), \texttt{signal}(c), \texttt{empty}(c), \texttt{sigall}(c)

  – \texttt{wait}(m, c): < V(m); wait(c)>; P(m)
    • make sure no interference is possible immediately after the \texttt{V(m)}
    • this corresponds to the “signal-and-continue” discipline

• Assumption: exclusion is guaranteed by (the context of) the application
  – operation called only from within critical sections
  – no (additional) exclusion is needed on the data structure associated with \texttt{c}
  – the implementation might store a reference to the guarding semaphore with \texttt{c} in order to check this
Example: implementing wait/signal

Wait \((m, c)\):

- EnterCS; \{ inhibit interrupts; and use \(m \rightarrow lock\) (spinlock) for this semaphore \}
- \(m \rightarrow val := m \rightarrow val +1\);
- \(\text{if } m \rightarrow val \leq 0 \text{ then}
  \begin{align*}
  &\text{Release one from } m \rightarrow \text{queue}; \\
  &\text{fi;}
  \end{align*}
- \(\text{Store self in } c \rightarrow \text{queue};\)
- \(\text{Call Scheduler; } \{ \text{stop current (putting it in blocked queue!)}), \text{LeaveCS, start other; } \}
- \text{EnterCS; } \{ \text{inhibit interrupts; and use } m \rightarrow lock \text{ (spinlock) for this semaphore } \}
- \(m \rightarrow val := m \rightarrow val -1;\)
- \(\text{if } m \rightarrow val < 0 \text{ then}
  \begin{align*}
  &\text{Store self in } m \rightarrow \text{queue}; \\
  &\text{Call Scheduler; } \{ \text{stop current (putting it in blocked queue!)}), \text{LeaveCS, start other; } \}
  \end{align*}
- \text{else LeaveCS } \{ \text{Release } s \rightarrow lock \text{ and enable interrupts} \}
- \text{fi}

Signal\((c)\):

- \(\text{if not empty}(c \rightarrow \text{queue}) \text{ then}
  \begin{align*}
  &\text{Release one from } c \rightarrow \text{queue;}
  \end{align*}
- \text{fi}

- signal uses no exclusion as there can be no other process accessing \(c\)
- manipulating the process records in \text{Scheduler} and \text{Release} needs protection
Agenda

• What is required to implement a kernel?
• General system calls
• Scheduling
Scheduling (allocation)

• **Resource scheduling** (allocation): assignment of resources to tasks

  – Schedule $S$ is a function that maps a time and a resource to a task:
    $S(t,r) = P$ means that task $P$ is assigned resource $r$ at time $t$

  – What a task is, is context dependent
    • e.g. a process, the reading of a disk block, handling an interrupt
    • .... a thread of execution, an activity

  – The *processor resource* is a special case
    • we usually say that *the task is running*

  – The interesting part of a schedule is when there is a change
    • the decision *procedures* for change are interesting
    • also when decisions are possible (system is in decision *mode*)
      – when is this mode reached for e.g. the processor resource or a memory segment?
        » processor: e.g. process into ready queue; end of time slice; process yielding (e.g. blocked)
        » memory: e.g. memory management call; replacement policy by memory subsystem; process termination

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking
Example schedules

- $P_0$, $P_1$, $P_2$: tasks
- $t_0$, $t_1$,....: scheduling points
  - system is in decision mode
  - scheduling decisions are taken
  - scheduling points may differ per resource (not shown here)

- processor resource: $PROC$
  - $S([t_0..t_1), PROC) = P_0$, $S([t_1..t_2), PROC) = P_1$, etc.
- a memory page: $m_{23}$
  - $S([t_0..t_4), m_{23}) = P_0$, etc.
- another memory page: $m_{56}$
  - $S([t_3..t_4), m_{56}) = P_2$, etc.
Scheduling policies and mechanisms

- The scheduling policy represents the strategy for allocating a resource to a task while in decision mode

  - policy: informal algorithm, tells the decision based on scheduling criteria.
    - task attributes (deadline, response time, ...)
    - the current state, represented by sets of ready processes as well as available and required resources
  - ...or on a lookup
    - pre-computed table

- Scheduling (allocation) mechanisms are different per resource type
  - processor:
    - implicit (i.e., not visible in the program text) management at scheduling points by the OS
      - through saving and restoring context
  - passive resource:
    - explicit locking and unlocking through e.g. semaphores (with FIFO policy) or implicit upon function entry and exit [monitors, e.g. Java]
    - implicit management through the OS, e.g. for memory
Policy: first (last) come first serve

• Resource assigned to tasks in (reverse) order of request arrival
  – resource held until task releases them
  – FCFS results in non-preemptable resource allocation
    • only resource scheduling at task synchronization points
    • example: printer spooling

• Applicable to all resource types
  – but typically for state-holding, non-preemptable resources
  – ... because for those, preemption means a loss of work
Policy: time sliced

• Time sliced: resource is given to a task for a certain amount of time (the quantum)
  – needs preemptable resources
    • e.g. processor
  – use in combination with other policies
    • round robin = time sliced + fcfs (with partial served sent back to queue)
    • typically for a processor

• Other policies
  – Real-time policies
    • earliest deadline first
    • rate monotonic
  – Shortest job next
    • Printer spooling
    • Minimizes the metric: average waiting time
Task attributes, metrics

- **Attribute**: a property the task has, static or dynamic
- **Metric**: *observed* property, result of applying scheduling policy

- **Attributes for decision making**
  - Arrival time (earliest service time) of a task
  - Computation time
  - Resource needs (e.g. memory)
  - Deadline

- **Metrics, defining the quality of the resulting behavior**
  - **utilization**
    - fraction of time the resource is used
  - **response time**
    - most texts: time elapsing from arrival to completion
      - in our book this is called: *turnaround time*
    - our book: time elapsing from arrival to first response
  - maximum or average values of these responses and turnaround times
  - throughput, number of deadline misses, average ‘overshoot’ etc.
Task time attributes in a picture

- A task has
  - a name (the $j^{th}$ task)
  - a (worst case) execution time $S_j$
  - a period, sometimes
  - a relative deadline, sometimes

- Dynamically, we can name the $i^{th}$ instance or occurrence
  - an arrival time (or earliest-start-time) $a_{j,i}$
  - an absolute deadline, sometimes (add $D_j$ to arrival time) $dl_{j,i}$
  - a start time (or beginning time) $b_{j,i}$
  - a departure time, or end time $e_{j,i}$

- (book) Execution time (between i/o): burst
- (book) Response time: $b_{j,i} - a_{j,i}$
- (book) Turnaround time: $e_{j,i} - a_{j,i}$
Policy implementation

- Communication between application and OS
- Preemption
- Data structures
- Decision mode
Policy implementation for the processor resource

- Compute a table beforehand, offline
  - static, specialist

- Compute the next assignment dynamically
  - order derived from task-set properties
  - requires knowledge of the tasks
    - possible for e.g. disk access, but more difficult for general processes
    - may try prediction

- **Common approach**: use priorities as an intermediate between application and OS
  - the next task is the one with the highest priority
    - break ties using an arbitration rule
  - the priority assignment policy is with the application

- Priorities are
  - explicitly assigned, by the programmer or
  - computed, through a function that depends on task properties, e.g.
    - memory use, timing: duration (estimated), deadline, period
    - i.e., the mentioned scheduling policies
Resources and preemption

• A task using a resource usually generates associated state
  – registers etc. in a processor
  – variables inside an object
  – ... what about a cache? is there state?
  – ... are there state-less resources?

• Upon preemption (due to priority or time slice expiration)
  – save the state
  – or destroy the state – roll-back
    • destroys the effort (work) in obtaining it
    • is a network interface card preemptable? or a printer?
  – associated penalty: context-switch time (of that resource)

• Preemption not always (directly) possible
  – hold the resource until preemption is possible (or operation finished)
  – leads to cooperative scheduling
  – penalty for waiters: incurred blocking time (perhaps even deadlock) and priority inversion
Priority inversion

- A low priority task obtains a resource; a high priority task waits on it
- A middle priority task pre-empts the low priority task
  - the high priority task now waits on the middle priority task
  - ... and executes effectively at the low priority
  - ... comparable to unfairness in synchronization
A solution: priority inheritance protocol

- The priority of the task $P$ using the resource is *dynamically adjusted* to be the maximum of
  - the priority of any other task that is blocked on the allocated resources of $P$
  - (... and its own priority)
- ...middle priority jobs will wait now.
Process scheduling: decision mode

• Process executes a blocking kernel call
  – e.g. a disk read of a block currently not in memory
  – the implementation of the call will store the process state at the reason of the blocking

• A timer interrupt occurs
  – the timer service routine may cause switching processes and use the information saved upon interrupt for this purpose
    • store context of interrupted process

• The process yields the processor
  – i.e. a ‘yield’ call
  – or after a call that changes a process state (e.g. `fork()`, `exit()` or communication), rescheduling may be needed
Actions in process scheduling and switching

- Executed somewhere in a “return path”, from kernel to user mode
  - as conclusion of a system call
    - blocking system call
    - process management call
  - as conclusion of an interrupt routine (typically, timer)

- In return path: determine process(es) that should run, according to policy

- For running processes (if any, and if they must be descheduled)
  - interrupt processor, which results in
    - store state vector
    - update accounting information
    - place in ready queue

- Select and start processes
  - load state vector into registers, prepare stack image
  - place process into running queue (which is just a register, for single processor)

- **Note**: last two steps: synchronization problem for multiple processors
Exercises

• **K.1** Implement $P(s)$, $V(s)$ for a binary semaphore $s$ using
  – a. *Fetch&Add*
  – b. *Compare&Swap*

• **K.3** A read system call initiates the device driver to read the required block from disk into a kernel buffer. The system call subsequently yields the processor in favor of another process. The disk interrupt handler is called upon completion of writing the block.
  – Is it possible for the handler to write the block into the memory space of the requesting process? When is this possible and when not? Explain your answer carefully.
Exercises

- **K.2.** Give an implementation of
  - signal-and-exit
  - signal-and-wait
  - signal-and-continue

using just semaphores.