ID1217 Concurrent Programming
Lecture 14

ROYAL INSTITUTE OF TECHNOLOGY

Implementations of Processes, Semaphores and Monitors

Vladimir Vlassov
KTH/ICT/EECS

ROYAL INSTITUTE OF TECHNOLOGY

Outline

- A single processor (SP) kernel
  - Structure of the kernel
  - Data structures and primitives – system calls
  - Outline of SP kernel
  - Semaphores in the kernel
- Process synchronization by the kernel
  - Implementing semaphores.
  - Implementing monitors.
- A multiprocessor (MP) kernel
  - Changes to the SP kernel
  - Outline of MP kernel
  - MP locking principle

ROYAL INSTITUTE OF TECHNOLOGY

Review: Shared Memory Programming

- Considered, so far: shared memory programming model
  - Joinable and detached threads interact via shared variables
  - Need a SW mechanism (with a HW support) to create, quit, join, schedule and synchronize threads
- Synchronization:
  - Mutual exclusion **`<S;>`**
  - Condition synchronization:

    ```
    <await (B) S;>
    <await (B)>
    ```
  - Busy waiting (locks, barriers) versus blocking (semaphores, condition variables)

Review: Busy Waiting Synchronization Mechanisms

- Locks – for mutual exclusion
  - Relies on atomic HW instructions, e.g. swap, test-and-set, fetch-and increment.
  - Unfair spin locks (e.g. test-and-set, test-test-and-set locks)
  - Fair (queuing) locks (e.g. the ticket lock, the bakery lock)
- Barriers – for collective synchronization (wait for all)
  - Busy waiting with locks, counters and signaling flags.
  - Block waiting with locks and condition variables (or semaphores).
  - Centralized barriers (e.g. counter and coordinator)
  - Decentralized (e.g. combining tree, butterfly, dissemination)

Review: Blocking Synchronization Mechanisms

- Condition variables
  - Queues of suspended processes waiting to be resumed (when some condition becomes true)
  - A signaling mechanism to suspend/resume processes holding locks
    - Blocking wait, Signal to resume
- Semaphores
  - Shared nonnegative counters or binary flags
  - P (pass) and V (release) atomic operations; P is blocking
  - Can be implemented in SW using locks and condition variables
  - Low memory demand per semaphore but needs memory for the delay queue

Review: Monitors

- Monitors
  - Abstract data types with mutual exclusive (atomic) operations
  - Use implicit locks (semaphores) for mutual exclusion
  - Used explicit (implicit in Java) condition variables for condition synchronization as a signaling mechanism
  - Both monitor locks and condition variables have blocking semantics
    - Entry queue and condition variable queues – a process can be in either of them, but only in one

Review: Overhead

- Process creation, joining, termination, scheduling
- Context switch
  - Save /restore a process state
  - How much memory to save the state and how fast (latency)
    - Make threads resident in CPU
- Synchronization
  - Memory demand: shared and private variables used for synchronization (counters, flags, queues, lists, etc.)
  - Latency: Transfer of locks, signals, maintenance of barriers, lists, queues, waiting in queues
    - Context switching overhead can be included here

ROYAL INSTITUTE OF TECHNOLOGY

Implementation of Processes

- We need a hardware-supported software mechanism that allows creation, termination, scheduling and synchronization of processes (threads)
- Consider outlines of
  - A Single-processor kernel
    - To implement concurrent processes on a single processor
  - A Multiprocessor kernel
    - To implement parallel and concurrent processes on a multiprocessor

Creating and Terminating Processes

ROYAL INSTITUTE OF TECHNOLOGY

- Consider spawning of concurrent threads:

  $\mathtt{S_0;}$

  $\mathtt{co\ P_1{:}\ S_1;\ ||\ P_2{:}\ S_2;\ ||\ \ldots\ ||\ P_n{:}\ S_n;\ oc}$

  $\mathtt{S_{n+1};}$

  - Here: $\mathtt{P_i}$ – a process name, $\mathtt{S_i}$ – process body (statements and variables)
- A parent proc spawns a child proc by the "system call" **`fork`** and joins a child proc by the "system call" **`join`**:

  $\mathtt{S_0;}$

  $\mathtt{for\ [i = 1\ to\ n]\ fork(P_i);}$

  $\mathtt{for\ [i = 1\ to\ n]\ join(P_i);}$

  $\mathtt{S_{n+1}}$
- To terminate, each child process performs a "system call" **`quit`**:

  $\mathtt{S_i;\ quit();}$
- The"system calls", fork, join and quit, are served (executed) in a kernel.

A Kernel

- A ***kernel*** (a.k.a. nucleus) – a set of data structures and subroutines that are at the core of any concurrent program.
  - Provides general functionality for any program: process management (create, join, quit, schedule) and synchronization (e.g. semaphores)
  - Kernel's data structures:
    - Descriptors – for processes, semaphores, etc.
    - Lists – free descriptors, ready processes, blocked processes, etc.
  - Kernel's subroutines (components):
    - Interrupt handlers – to handle system calls (software interrupts) and HW interrupts
    - Kernel primitives – atomic operations on kernel data structures
    - Dispatcher (scheduler)

A Single Processor (SP) Kernel

- First consider an outline of a SP kernel:
  - Implements processes and synchronization on a single processor
  - Consider only relevant functionality: basic process management and blocking synchronization, e.g. semaphores and monitors
  - Not covered: dynamic storage allocation, priority scheduling, i/o operations, etc.
- A processor executes either a process routine or a kernel routine (a.k.a. primitive)
  - Switch from the process context to the kernel – by a software interrupt (trap) – a special machine instruction executed by the calling process

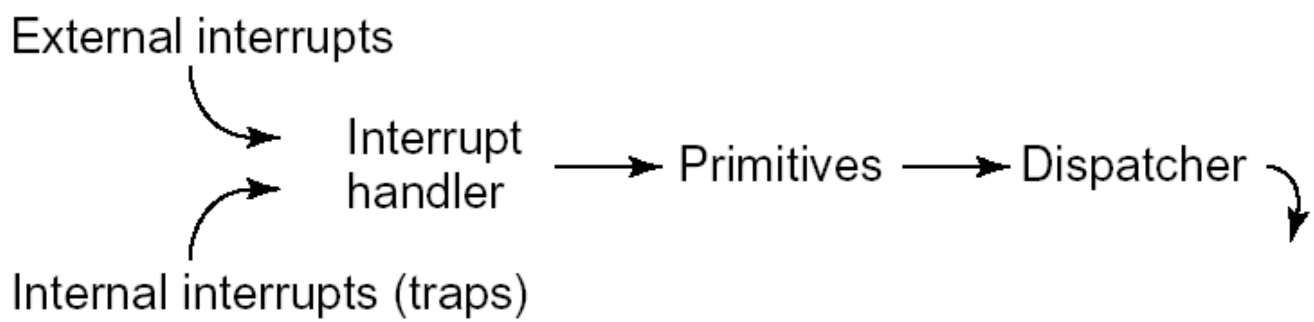
ROYAL INSTITUTE OF TECHNOLOGY

SP Kernel's Data

- ***Process descriptors*** – to keep track of processes
  - A process descriptor can be in one of the following lists
- Kernel lists:
  - The ***free list*** – empty descriptors
  - The ***ready list*** – descriptors of processes ready for execution
  - The ***waiting list*** – descriptors of processes waiting for join
- **`executing`** – a variable that indicates a currently executing process or a process to be executed
  - If zero – no process is executing – a signal to the dispatcher to activate a ready process if any

Kernel Components and Control Flow

- ***Interrupt handlers*** – routines executed on external or internal interrupts (traps)
  - E.g. the SVC handler and the handler of interrupts from timer
- ***Primitives*** (routines) e.g. fork, join, quit
- ***Dispatcher*** – a routine that performs context switching



Interrupt Handlers

- ***Timer handler*** – accepts external interrupts from a HW interval timer
    - The HW interval timer is loaded with a positive integer value, counts down with a fixed rate and signals an interrupt when it reaches zero.
  - For time slicing: allows swapping processes
  - The handler sets executing to zero and calls dispatcher for a context switch
- ***SVC handler*** – Supervisor Call handler – accepts internal software interrupts (traps to the kernel)
  - Saves the state of the executing process– a context switch to the kernel.
  - Decodes the "system call" and invokes an appropriate kernel primitive.
- Interrupt handlers are entered with interrupt inhibited – for atomicity
- There should be more interrupt handlers – not considered here.

Primitives for Process Management

- **`fork(`**initial process state**`)`**
  - creates a process (allocates descriptor and a context) and makes it eligible for execution;
  - Arguments: the address of the first instruction, and other data representing process initial state.
- **`quit(`**status**`)`**
  - terminates the executing process
- **`join(`**process id**`)`** – blocking call
  - waits for a specified process to terminate
  - If no process is specified, wait for any child process to terminate

Outline of a Single-Processor Kernel

- Variables and Handlers

```
processType processDescriptor[maxProcs];
int executing = 0;     # index of the executing process
declarations of variables for the free, ready, and waiting lists;

SVC_Handler: {    # entered with interrupts inhibited
  save state of executing;
  determine which primitive was invoked, then call it;
}

Timer_Handler: {    # entered with interrupts inhibited
  insert descriptor of executing at end of ready list;
  executing = 0;
  dispatcher();
}
```

Outline of a Single-Processor Kernel (cont'd)

- Process management primitives

```
procedure fork(initial process state) {
   remove a descriptor from the free list and initialize it;
   insert the descriptor on the end of the ready list;
   dispatcher();
}
procedure quit() {
   record that executing has quit;
   insert descriptor of executing at end of free list;
   executing = 0;
   if (parent process is waiting for this child) {
      remove parent from the waiting list; put parent on the ready list; }
   dispatcher();
}
procedure join(name of child process) {
   if (child has not yet quit) {
      put the descriptor of executing on the waiting list;
      executing = 0;
   }
   dispatcher();
}
```

Outline of a Single-Processor Kernel (cont'd)

- The dispatcher routine:
  - Invoked after any primitive and on timer interrupts
  - A context switch is requested by setting **`executing`** to zero
  - If **`executing`** is zero, the dispatcher activates a process (if any) from front of the ready list
  - If the ready list is empty, activates the "idle" process
  - When activates a process, enables interrupts.

```
procedure dispatcher() {
  if (executing == 0) { # current process blocked or quit
    remove descriptor from front of ready list;
    set executing to point to it;
  }
  start the interval timer;
  load state of executing;    # with interrupts enabled
}
```

Adding Semaphores to the Kernel

- A semaphore descriptor points to a record:
  - A unique name
  - A variables to hold a value of the semaphore
  - A list (queue) of blocked processes waiting for the semaphore to become positive
- A process descriptor can be on one of the lists:
  - ready,
  - waiting for join,
  - empty descriptors,
  - semaphore wait lists.

Semaphore Primitives in the Kernel

- **`createSem(value, name)`**
  - Create a semaphore with a name and initialize with a given value
- **`Psem(name)`**
  - P operation – decrement when positive
  - May cause a context switch – The calling process is delayed if sem <= 0
- **`Vsem(name)`**
  - V operation – increment
  - Does not cause a context switch – The calling process continues
  - A waiting process (if any) is resumed
- Mutual exclusion is guaranteed by atomicity of a kernel primitive
- Semaphores should be destroyed when no longer needed
  - Add a special **`destroySem`** primitive to the kernel

KTH VETENSKAP OCH KONST

ROYAL INSTITUTE OF TECHNOLOGY

Semaphore Primitives in the SP Kernel.

```
procedure createSem(initial value, int *name) {
   get an empty semaphore descriptor;
   initialize the descriptor;
   set name to the name (index) of the descriptor;
   dispatcher();
}
procedure Psem(name) {
   find semaphore descriptor of name;
   if (value > 0)
     value = value - 1;
   else {
     insert descriptor of executing at end of blocked list;
     executing = 0;    # indicate executing is blocked
   }
   dispatcher();
}
procedure Vsem(name) {
   find semaphore descriptor of name;
   if (blocked list empty)
     value = value + 1;
   else {
     remove process descriptor from front of blocked list;
     insert the descriptor at end of ready list;
   }
   dispatcher();
}
```

Adding Monitors and Condition Variables to the Kernel

- The kernel can provide the following services for processes that use monitors and condition variables
  - Control monitor locks
  - Maintain entry queues of processes waiting for the locks
  - Delay a process on a condition variable
  - Maintain queues of processes waiting on condition variables
  - Signal a process waiting on a condition variable to resume
  - Monitors and condition variables must be uniquely identified in the kernel by descriptors

ROYAL INSTITUTE OF TECHNOLOGY

Kernel's Primitives for Monitors and Condition Variables

- **`enter(monitor)`**
  - Enter a specified monitor.
  - A calling process is delayed if the monitor entry is locked.
- **`exit(monitor)`**
  - Causes a process waiting in the monitor's entry queue to resume.
  - The calling process continues.
- **`wait(monitor, cond)`**
  - Delay the calling process on a specified condition variable and release a specified monitor lock.
  - Causes a context switch.
- **`signal(monitor, cond)`**
  - Move a process from the cond var queue to the monitor entry queue of the
  - The calling process continues.

Monitor Primitives in the SP Kernel

```
procedure enter(int mName) {
   find descriptor for monitor mName;
   if (mLock == 1) {
      insert descriptor of executing at end of entry queue;
      executing = 0;
   }
   else
      mLock = 1;     # acquire exclusive access to mName
   dispatcher();
}
procedure exit(int mName) {
   find descriptor for monitor mName;
   if (entry queue not empty)
      move process from front of entry queue to rear of ready list;
   else
      mLock = 0;     # clear the lock
   dispatcher();
}
procedure wait(int mName; int cName) {
   find descriptor for condition variable cName;
   insert descriptor of executing at end of delay queue of cName;
   executing = 0;
   exit(mName);
}
procedure signal(int mName; int cName) {
   find descriptor for monitor mName;
   find descriptor for condition variable cName;
   if (delay queue not empty)
      move process from front of delay queue to rear of entry queue;
   dispatcher();
}
```

Multiprocessor Kernel

- Each processor can execute a different process
  - `int executing[n];` // array
- Kernel code can be executed by any processor
  - A process traps to a local instance of the kernel on the processor where the process is executing
  - The kernel inhibits interrupts only on that processor
- Kernel data are shared among kernel instances
  - Must synchronize access to the kernel data
  - Critical data structures: free, ready, waiting lists – use fair critical section implementation
- Two major changes
  1. Need to modify dispatcher to make use of all processors
  2. Need to use locks to provide exclusive access to kernel data

Multiprocessor Locking Principle

- Make critical sections short
- Use separate locks for each data structure
  - E.g. separate locks for the free, ready, and waiting lists to be accessed in critical sections
- Need to be careful to avoid deadlock

Outline of a Shared-Memory Multiprocessor Kernel

```
processType processDescriptor[maxProcs];
int executing[maxProcs];    # one entry per processor
declarations of free, ready, and waiting lists and their locks;

SVC_Handler: {
  # entered with interrupts inhibited on processor i
  save state of executing[i];
  determine which primitive was invoked, then call it;
}

Timer_Handler: {
  # entered with interrupts inhibited on processor i
  lock ready list; insert executing[i] at end; unlock ready list;
  executing[i] = 0;
  dispatcher();
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Outline of a MP Kernel (cont'd)

```
procedure fork(initial process state) {
  lock free list; remove a descriptor; unlock free list;
  initialize the descriptor;
  lock ready list; insert descriptor at end; unlock ready list;
  dispatcher();
}

procedure quit() {
  lock free list; insert executing[i] at end; unlock free list;
  record that executing[i] has quit; executing[i] = 0;
  if (parent process is waiting) {
    lock waiting list; remove parent from that list; unlock waiting list;
    lock ready list; put parent on ready list; unlock ready list;
  }
  dispatcher();
}

procedure join(name of child process) {
  if (child has already quit)
    return;
  lock waiting list; put executing[i] on that list; unlock waiting list; executing[i] = 0;
  dispatcher();
}
```

Outline of a MP Kernel (cont'd)

```
procedure dispatcher() {
  if (executing[i] == 0) {
    lock ready list;
    if (ready list not empty) {
      remove descriptor from ready list;
      set executing[i] to point to it;
    }
    else     # ready list is empty
      set executing[i] to point to Idle process;
    unlock ready list;
  }
  if (executing[i] is not the Idle process)
    start timer on processor i;
  load state of executing[i];     # with interrupts enabled
}
```

Idle Processors

- Should get them busy when there are threads to execute
- Three strategies:
  1) Have each idle processor execute an "idle process“ that regularly examines the ready list
  2) Have each process executing **`fork`** search for an idle processor and assign a new process to it
  3) Use a separate dispatcher process executed on its own processor that schedules ready processes to idle processors
- The first strategy (idle process) is the most efficient
  – What else can an idle processor do?

Outline of the Idle Process

```
process Idle {
  while (executing[i] == the Idle process) {
    while (ready list empty) Delay;
    lock ready list;
    if (ready list not empty) {
      remove descriptor from front of ready list;
      set executing[i] to point to it;
    }
    unlock ready list;
  }
  start the interval timer on processor i;
  load state of executing[i];    # with interrupts enabled
}
```

Issues for NUMA

- Load balancing and data distribution are issues for distributed shared memory MP (NUMA)
  - Stationary processes versus migratory (mobile) processes
  - Migratory data
- A kernel instance on each processor can maintain its own local ready list of processes executing on that processor
  - A suspended process is resumed on the same processor where it was suspended
  - Cached data can be still in the cache up to date
  - TLB
  - Shared data and code should be in the local memory
- Co-scheduling
  - Common shared data are in the cache, TLB is common