Operating Systems: Processes and Threads

Week 1: Lecture 2, Thu 8.9.2011

Tiina Niklander
Process model
Process

- is an activity that has a program, input, output and a state.

- Some terms:
  - **Text/code** = executable instructions
  - **Data** = variables
  - **Stack** = work area
    - Parameter passing to subroutines/system calls
  - **Process Control Block, PCB entry in Process Table** = management information
Process

Sta Fig 3.12
Process model

- One physical program counter switches between processes
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant on one CPU
  pseudoparallelism
Process Creation

Principal events that cause process creation

1. System initialization
2. Execution of a process creation system
3. User request to create a new process
4. Initiation of a batch job
Process Creation

Operating system does
- Create PCB (new process table entry)
  - OS ‘generates’ a unique ID
- Allocate memory for the process
- Initiate PCB
- Link PCB to other structures
  - Place to Ready-queue, link to parent process, etc
Process Termination

Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)
Process Hierarchies

■ Parent creates a child process, child process can create its own process
■ Forms a hierarchy
  ■ UNIX calls this a "process group"
■ Windows has no concept of process hierarchy
  ■ all processes are created equal
Process States (1)

- Possible process states
  - running
  - blocked
  - ready
- Transitions between states shown

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
Process states

Figure 3.7 Process States for Trace of Figure 3.4
Process states

Sta Fig 3.8
## Process Control Block (prosessin kuvaaja)

<table>
<thead>
<tr>
<th><strong>Process management</strong></th>
<th><strong>Memory management</strong></th>
<th><strong>File management</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fields of a process table entry**
Tasks on interrupt

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

The process is not always changed in step 6, if running process can continue.
Process can be changed only at step 6.
Threads
Word processor with three threads

- One thread interacts with user
- One thread handles reformatting in background
- One thread handles disk backups in background
Multithreaded web server using dispatcher

- Dispatcher receives requests
- Handles to an idle worker thread for processing
- Suitable model for such: finite-state machine

```c
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

```c
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```
Thread model

(a) Three processes each with one thread
(b) One process with three threads
# Thread Model: Process vs thread

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

- Items shared by all threads in a process
- Items private to each thread in the thread
- Each thread has its own stack!
Why threads?

- Multiple parallel activities. Threads provide easier programming models
- Faster to create and destroy one thread than a whole process
- When one thread waits, another might be able to execute
- Shared data area of threads in one process. Efficient resource sharing and easy communication.

**BUT**

- Mutual exclusion and synchronization are fully programmer’s responsibility (no support from OS or anything else)
POSIX threads (pthreads)

IEEE standard for threads for portable thread programs

- pthread_create()
  - Create a new thread
- pthread_exit()
  - Terminate the calling thread
- pthread_join()
  - Wait for a specific thread to exit
- pthread_yield()
  - Release the CPU to let another thread run
- Functions for synchronization and mutual exclusion
- And more than 50 other functions
(a) A user-level threads package.

- Kernel (or OS) is not aware of threads, schedules processes
- User process must dispatch threads itself

(b) A threads package managed by the kernel.

- All thread control and dispatching done by the kernel
- No control on the user level
User-level threads

**Advantages**
- Fast dispatching
  - No mode switch
  - No interrupt
  - No process switch!
- Programmer can freely choose the scheduling mechanism

**Disadvantages**
- When one thread is blocked on system call, it blocks the whole process (and all other threads)
- Threads of one process cannot be executed on several processors or cores concurrently
  - Remember: kernel dispatches only processes!
Kernel-level threads

**Advantages**
- Threads of one process can be executed simultaneously on multiple processors
- If one thread is Blocked, the other threads of this process may still continue
- Often also the kernel implementation is multithreaded

**Disadvantages**
- Dispatching a thread has two phases:
  - Interrupt +, interrupt handling and mode switch
  - Dispatcher (and return to user mode)
- Slower than in user-level threads
Hybrid Implementations

Multiplexing user-level threads onto kernel-level threads

Diagram:
- Multiple user threads on a kernel thread
- Kernel space
- User space

25
Figure 4.15  Solaris Multithreaded Architecture Example
Making single-threaded code multithreaded

- Global variables
  - Threads may not be aware of others using the same variable
  - See next slide for solutions

- Library procedures
  - Might not be reentrant (second call to procedure before the first one is finished is not safe)
  - Solution: rewrite the library or use an excluding jacket

- Signals
  - No simple solutions – difficult already in single thread

- Stack management
  - How to increase a thread’s stack in case of stack overflow
Threads and global variables

Conflicts between threads over the use of a global variable

One solution:
- Prohibit global variables

Alternative:
- Private global variables
- Accessing these is tricky, language may not support
- New library procedures
Interprocess Communication
Race Conditions

Two processes want to access shared memory at same time
Critical Regions (1)

Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region
Critical Regions (2)

Mutual exclusion using critical regions
Mutual Exclusion

- Busy Waiting – occupy the CPU while waiting
- Sleep and Wakeup
  - Semaphores
  - Mutexes
  - Monitors
  - Message Passing
  - Barriers

Concurrent Programming
Rinnakkaisohjelmointi
Bursts of CPU usage alternate with periods of I/O wait
- a CPU-bound process
- an I/O bound process
Introduction to Scheduling (2)

Scheduling Algorithm Goals

All systems
- Fairness - giving each process a fair share of the CPU
- Policy enforcement - seeing that stated policy is carried out
- Balance - keeping all parts of the system busy

Batch systems
- Throughput - maximize jobs per hour
- Turnaround time - minimize time between submission and termination
- CPU utilization - keep the CPU busy all the time

Interactive systems
- Response time - respond to requests quickly
- Proportionality - meet users’ expectations

Real-time systems
- Meeting deadlines - avoid losing data
- Predictability - avoid quality degradation in multimedia systems
An example of shortest job first scheduling
Scheduling in Batch Systems (2)

Three level scheduling
Scheduling in Interactive Systems (1)

- Round Robin Scheduling
  - a) list of runnable processes
  - b) list of runnable processes after B uses up its quantum
A scheduling algorithm with four priority classes
Scheduling in Real-Time Systems

Schedulable real-time system

- Given
  - \( m \) periodic events
  - Event \( i \) occurs within period \( P_i \) and requires \( C_i \) seconds
- Then the load can only be handled if

\[
\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1
\]
Policy versus Mechanism

- Separate what is **allowed** to be done with **how** it is done
  - a process knows which of its children threads are important and need priority

- Scheduling algorithm parameterized
  - mechanism in the kernel

- Parameters filled in by user processes
  - policy set by user process
Thread Scheduling (1)

Possible scheduling of threads

- 50-msec process quantum
- threads run 5 msec/ CPU burst

User-level

Kernel-level
Extra pages:
Scheduling example
CPU Scheduling: Algorithms examples

- First-Come-First-Served (FCFS)
- Round Robin (RR)
- Virtual Round Robin (VRR)
- Shortest Process Next (SPN)
- Shortest Remaining Time (SRT)
- Highest Response Ratio Next (HRRN)
- Multilevel Feedback (Feedback)
- Fair Share Scheduling (FSS)
Process set

Service Time = running time on CPU
No I/O in the example (unrealistic?)

(Tbl 9.4 [Stal05])
FCFS - First Come First Served

(Fig 9.5 [Stal05])

- Arrival time for B: 4 (avg.)
- Completion time for B: 12
- CPU time for B: 12
- Avg. completion time for B: 8.6

R
3 7 9 12 12

2 5 4 6 3 2
RR - Round Robin

(Fig 9.5 [Stal05])

avg. 10.8
RR - effect of the time slice

(Fig 9.5 [Stal05])
Figure 9.7  Queuing Diagram for Virtual Round-Robin Scheduler
SPN - Shortest Process Next

(Fig 9.5 [Stal05])

avg. 7.6
SRT - Shortest Remaining Time

(Fig 9.5 [Stal05])

avg. 7.2
**HRRN - Highest Response Ratio Next**

Time spent waiting CPU + expected service time

Response ratio = \[
\frac{(5+2)}{2} > \frac{(7+5)}{5}
\]

(Fig 9.5 [Stal05])

Average 8.0
Multilevel Feedback

(Fig 9.10 [Stal05])
Feedback \( q=1 \)

(Fig 9.5 [Stal05])

4 (avg.)

mistake?
Feedback q=2i

(Fig 9.5 [Stal05])

keskim. 10.6

mistake?
## Comparation

<table>
<thead>
<tr>
<th>Selection Function</th>
<th>Decision Mode</th>
<th>Throughput</th>
<th>Response Time</th>
<th>Overhead</th>
<th>Effect on Processes</th>
<th>Starvation</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCFS</td>
<td>max[w]</td>
<td>Nonpreemptive</td>
<td>Not emphasized</td>
<td>May be high, especially if there is a large variance in process execution times</td>
<td>Minimum</td>
<td>Penalizes short processes; penalizes I/O bound processes</td>
</tr>
<tr>
<td>Round Robin</td>
<td>constant</td>
<td>Preemptive (at time quantum)</td>
<td>May be low if quantum is too small</td>
<td>Provides good response time for short processes</td>
<td>Minimum</td>
<td>Fair treatment</td>
</tr>
<tr>
<td>SPN</td>
<td>min[s]</td>
<td>Nonpreemptive</td>
<td>High</td>
<td>Provides good response time for short processes</td>
<td>Can be high</td>
<td>Penalizes long processes</td>
</tr>
<tr>
<td>SRT</td>
<td>min[s – e]</td>
<td>Preemptive (at arrival)</td>
<td>High</td>
<td>Provides good response time</td>
<td>Can be high</td>
<td>Penalizes long processes</td>
</tr>
<tr>
<td>HRRN</td>
<td>max((\frac{w+s}{s}))</td>
<td>Nonpreemptive</td>
<td>High</td>
<td>Provides good response time</td>
<td>Can be high</td>
<td>Good balance</td>
</tr>
<tr>
<td>Feedback</td>
<td>(see text)</td>
<td>Preemptive (at time quantum)</td>
<td>Not emphasized</td>
<td>Not emphasized</td>
<td>Can be high</td>
<td>May favor I/O bound processes</td>
</tr>
</tbody>
</table>

\(w\) = time spent in system so far, waiting and executing
\(e\) = time spent in execution so far
\(s\) = total service time required by the process, including \(e\)