CSE 120: Principles of Operating Systems

Lecture 2

Processes

September 30, 2003

Prof. Joe Pasquale

Department of Computer Science and Engineering
University of California, San Diego
Before We Begin ...

Read Chapters 4 and 5 (on Processes and Threads)
  • for background and overview, read Chapters 1-3

Discussion section: Wednesdays 12-1, HSS 1330
Introduction

Most fundamental kernel function: run a program

Users want ability to run multiple programs at once

How is this achieved given single CPU and memory?
What is a Process?

Abstraction of a running program

• “a program in execution”

Dynamic

• has state, changes over time
• whereas a program is static

Basic operations

• start/end, suspend/resume, send/receive messages
Goal: Support Multiple Processes

Users would like multiple programs running at same time
- some are more important, foreground/background
- not all actively using the CPU
- some waiting for input, devices (e.g., disk), ...

How to do this given single CPU (or small number)?
**Multiprogramming**

Given a running process

- at some point, it needs a resource, e.g., I/O device
- say resource is busy, process can’t proceed
- so, “voluntarily” gives up CPU to another process

**Yield (p)**

- let process p run (voluntarily give up CPU to p)
- requires context switching
Process Memory Structure

Text
- code: program instructions

Data
- global variables
- heap (dynamic allocation)

Stack
- activation records
- automatic growth/shrinkage
Process Stack

Stack of activation records
  • one per pending procedure

Each activation record stores
  • where to return to
  • link to previous record
  • automatic (local) variables

Stack pointer points to last record
  • RETURN instruction relies on it
Context Switching

Allocating CPU to a process requires context switching

• first, save context of currently running process
• next, load context of next process to run

Loading the context

• load general registers, stack pointer, etc.
• load the program counter (must be last instruction)
**Simple Context Switching**

Two processes: $P_1$ and $P_2$

$P_1$ calls yield () to voluntarily give up CPU to $P_2$

Save and restore registers
- general-purpose, stack pointer, program counter

Switch text and data (not necessary if shared: threads)

Switch stacks: note that PC is in the middle of yield ()!
The Magic of yield ()

magic = 0

Save P₁’s context
  • GP (general purpose) registers, SP (stack pointer)
  • Lastly, PC (program counter); note, inside yield!

If (magic == 1) return, else magic = 1

Restore P₂’s context
  • GP registers, SP
  • Lastly, restore PC
In this example, \( P_1 \) is about to set \( x \) to 7 and yield to \( P_2 \). \( P_2 \) had already yielded to \( P_1 \): note \( P_2 \)'s saved PC and SP.

Not shown are declarations: \( x \) is a global variable (in each process), and magic is a local variable in yield().

return to previous

magic: 1 (why is this 1?)
$P_1$ has just set $x$ to 7 and is about to call `yield`. The PC always points to the instruction to be executed next.
Upon entering yield, an activation record is pushed on the stack. It contains links, and local variable magic.

```
main ()
{ x = 7
  yield ()
  ...
}
yield ()
{ magic = 0
  save p1.context
  if magic == 1 ret
  else magic = 1
  restore p2.context
}
x: 7

p1.context: ...
PC
SP

P_1

PC
SP

shared memory

P_2

main ()
{ x = 11
  yield ()
  ...
}
yield ()
{ magic = 0
  save p2.context
  if magic == 1 ret
  else magic = 1
  restore p1.context
}
x: 11

p2.context: ...
PC
SP

P_2

PC
SP

shared memory

P_1

return to previous
magic: ?

return to previous
magic: 1

stack

stack

X: 7

X: 11

return to previous
magic: 1

return to previous
magic: ?

stack

stack

shared memory

14
Magic, an automatic variable because it is dynamically allocated on the stack, is set to 0. Next: save context.

P₁

main ()
{text}
{ x = 7
  yield ()
  ...
} yield ()
{ magic = 0
  save p1.context
  if magic == 1 ret
  else magic = 1
  restore p2.context
}
x: 7 data

shared memory

p1.context:
{...}
PC
SP

p2.context:
{...}
PC
SP

return to
stack

previous

magic: 0

P₂

main ()
text
{x = 11
  yield ()
  ...
} yield ()
{ magic = 0
  save p2.context
  if magic == 1 ret
  else magic = 1
  restore p1.context
}
x: 11 data

shared memory

p1.context:
{...}
PC
SP

p2.context:
{...}
PC
SP

return to
stack

previous

magic: 1
$P_1$'s context is now saved. The saved PC points just after the save context. Compare this to $P_2$'s saved context.
P₁ just checked whether magic equals 1, which was false, and so, on to the else clause to set magic to 1.

P₁

```plaintext
main () { x = 7
  yield ()
...}
yield () { magic = 0
  save p1.context
  if magic == 1 ret
  else magic = 1
  restore p2.context
}
x: 7

return to previous
magic: 0
```

P₂

```plaintext
main () { x = 11
  yield ()
...}
yield () { magic = 0
  save p2.context
  if magic == 1 ret
  else magic = 1
  restore p1.context
}
x: 11

return to previous
magic: 1
```
$P_1$ sets magic to 1, and is about to restore $P_2$’s context (just like $P_2$’s situation when it restored $P_1$’s context).
$P_2$'s context is restored (i.e., the machine state is now that of $P_2$). The PC points to the if statement.

```
main ()
{
    x = 7
    yield ()
    ...
    yield ()
    {
        magic = 0
        save p1.context
        if magic == 1 ret
        else magic = 1
        restore p2.context
    }
    x: 7
}
```

```
main ()
{
    x = 11
    yield ()
    ...
    yield ()
    {
        magic = 0
        save p2.context
        if magic == 1 ret
        else magic = 1
        restore p1.context
    }
    x: 11
}
Since magic equals 1, $P_2$ returns from yield (unlike last time when magic equaled 0). No wonder it’s called magic!

---

**P₁**

```plaintext
main ()
{ x = 7
  yield ()
  ...}
yield ()
{ magic = 0
  save p1.context
  if magic == 1 ret
   else magic = 1
  restore p2.context
}
x: 7
```

**P₂**

```plaintext
main ()
{ x = 11
  yield ()
  ...}
yield ()
{ magic = 0
  save p2.context
  if magic == 1 ret
   else magic = 1
  restore p1.context
}
x: 11
```

---

**Shared Memory**

- p1.context:
  ```plaintext
  ... PC
  SP
  ```
- p2.context:
  ```plaintext
  ... PC (these links are no longer valid)
  SP 
  ```
Another Way to Do This

Call causes adding activation record on P1’s stack
  • saves return location to point after yield () call

Save P1’s context: GP registers, Stack Pointer

Restore P2’s context: GP registers, Stack Pointer

Return
  • pop activation record, get return location
Timesharing: Illusion of Simultaneity

Multiple processes, single CPU
(on a uniprocessor)

Conceptually, each process makes progress over time

In reality, each periodically gets bit of CPU time: quantum

Illusion of parallel progress by rapidly switching CPU
How is Timesharing Implemented?

Kernel keeps track of progress of each process

Characterizes state of process’s progress

- **Running**: actually making progress (using CPU)
- **Ready**: able to make progress (not using CPU)
- **Blocked**: not able to make progress (not using CPU)

Kernel selects a ready process, lets it run

- eventually, the kernel gets back control
- selects another ready process to run, etc.
Process State Diagram

State transitions

- **Dispatch**: allocate the CPU to a process
- **Yield**: process voluntarily gives up CPU
- **Preempt**: forcibly take away CPU from process
- **Sleep**: process gives up CPU to wait for event
- **Wakeup**: event occurred, make process ready
Logical vs. Physical Execution

<table>
<thead>
<tr>
<th>Logical Execution</th>
<th>Able to execute</th>
<th>Not able to execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actually executing</td>
<td>Run</td>
<td>X</td>
</tr>
<tr>
<td>Not actually executing</td>
<td>Ready</td>
<td>Blocked</td>
</tr>
</tbody>
</table>
Process vs. Kernel

The kernel is the code that supports processes

- common system-related functions: system calls
  - fork(), exit(), wait(), read(), write(), ...
- system management functions
  - context switching, scheduling

When does the kernel run?

- whenever system call or hardware interrupt occurs

The kernel is not a process

- may be considered extension of last process to run
Processes and Kernel

Process 1
- Text
- Data
- Stack

Process 2
- Text
- Data
- Stack

... (for Process n)
- Text
- Data
- Stack

System Calls
- fork
- read
- write

Kernel
- Context Switching
- Scheduling
Kernel Maintains Process Table

List of processes and their states

- Process 1  Ready  (+ other state info)
- Process 2  Running  ...
- Process 3  Ready  ...
- Process 4  Blocked  ...

Other state info includes

- CPU context
- memory usage
- etc.
Context Switching by the Kernel

Process makes system call (TRAP) or is interrupted

Via hardware
• automatic mode switch occurs, user to kernel
• control goes to well-defined location in kernel

Via software (kernel)
• save context
• select new process from those that are ready
• restore context
• RTI: return from interrupt/trap
How Does Kernel Get Control

To allow a ready process to run, kernel must get control

Running process can give up control voluntarily
  • process makes a blocking system call, e.g., read()
  • or, process calls yield() to give up CPU
  • control goes to kernel, which dispatches a process

Or, processor is forcibly taken away (preemption)
  • while kernel is running, it sets a timer
  • when timer expires, interrupt is generated
  • hardware forces control to go to kernel
Protecting the Kernel from Processes

A process should not be able to modify the kernel.

Entry into the kernel must be controlled.

Requires support from hardware:

- two modes of operation: kernel and user
- processor status word: specifies mode

Kernel mode: kernel runs, all instructions available.

User mode: a process runs, instructions limited.
Processes vs. Threads

Single-threaded Processes

- process: “program in execution”
- one sequential path of execution (single thread)
- resources: memory (text, data, stack), etc.

Multi-threaded Processes

- process: contains one or more threads
- thread: one sequential execution path
- all of process’s threads execute in same memory
User-Level Threads

Process 1

Text

Data

Stack

Thread C/W + Sched

Process 2

Text

Data

Stack

Thread C/W + Sched

Process n

Text

Data

Stack

Thread C/W + Sched

Kernel

System Calls

fork
read
write

Process Context Switching

Process Scheduling
Kernel-Level Threads

Process 1

Text
Data
Stack 1
Stack 2
Stack 3

Process 2

Text
Data
Stack 1
Stack 2

Process n

Text
Data
Stack 1

System Calls
fork
read
write

Kernel
Thread Context Switching
Thread + Process Scheduling