Lecture 2

Processes

January 11, 2006

Prof. Joe Pasquale
Department of Computer Science and Engineering
University of California, San Diego

© 2006 by Joseph Pasquale
Before We Begin ...

Read Chapters 3 and 4 (on Processes and Threads)

Make sure you can log in to

- ieng9.ucsd.edu
- webboard.ucsd.edu

Reminder

- Homework is due on Friday at 5:00
- Programming project 1 due on Sunday at midnight
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
Two Basic Resources

To make progress, a process needs two basic resources

CPU (time, meaning cycles)
  • to execute instructions

Memory (space, meaning bytes or words)
  • to maintain state

Other resources
  • for I/O
Machine State of a Process

CPU or processor context
- PC (program counter)
- SP (stack pointer)
- GP (general purpose) registers

Memory
- Code
- Global Variables
- Stack of Activation Records

Other (other registers, memory, kernel-related state)
Process Memory Structure

Text
• **Code**: program instructions

Data
• **Static variables**, e.g., global
• **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
- Other (e.g., register values)

Stack pointer points to last record
- return instruction relies on it
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
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 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 $P_1$’s context

- GP (general purpose) registers, SP (stack pointer)
- lastly, PC (program counter) - note, inside yield!

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

restore $P_2$’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 \( \text{magic} \) is a local variable in \( \text{yield} () \). Can \( \text{magic} \) be global? Can it be shared?

---

\( P_1 \)

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

\( x: ? \)

---

\( P_2 \)

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

\( x: 11 \)

---

\( \text{shared memory} \)

\( \text{p1.context:} \)

```
...  
PC  
SP  
```

\( \text{p2.context:} \)

```
...  
PC  
SP  
```

---

\( \text{stack} \)

```
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.
Variable magic is set to 0. It is an "automatic" variable, dynamically allocated on the stack. Next: save context.

```
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
```

```
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
```
P₁’s context is now saved. The saved PC points just after the save context. Compare this to P₂’s saved context.
\( P_1 \) just checked whether magic equals 1, which was false, and so, on to the else clause to set magic to 1.

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

\[
\text{main ()}
\{
\text{\quad x = 11}
\quad \text{yield ()}
\}
\text{yield ()}
\{
\text{\quad magic = 0}
\quad \text{save p2.context}
\quad \text{if magic == 1 ret}
\quad \text{else magic = 1}
\quad \text{restore p1.context}
\}
\text{x: 11}
\]
P₁ sets magic to 1, and is about to restore P₂’s context (just like P₂’s situation when it restored P₁’s context).
P₂’s context is restored (i.e., the machine state is now that of P₂). The PC points to the if statement.
Since magic equals 1, $P_2$ returns from yield (unlike last time when magic equaled 0). No wonder it’s called magic!
Another Way to Do This

Call causes adding activation record on $P_1$’s stack

- Saves return location to point after yield () call

Save $P_1$’s context: GP registers, Stack Pointer

Restore $P_2$’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 quantum of CPU time
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, ...
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

### Logical Execution

<table>
<thead>
<tr>
<th>Able to execute</th>
<th>Not able to execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run</td>
<td>X</td>
</tr>
<tr>
<td>Ready</td>
<td>Blocked</td>
</tr>
</tbody>
</table>

### Physical Execution

- **Actually executing**
  - Run
  - Ready
- **Not actually executing**
  - X
  - Blocked
Process vs. Kernel

The kernel is the code that supports processes

- System calls: fork(), exit(), read(), write(), …
- System management: context switching, scheduling

When does the kernel run?

- When system call or hardware interrupt occurs

Is the kernel a process, i.e., a “program in execution”?

- No, it supports processes and devices
  - Runs as an extension of process making system call
  - Runs in response to device issuing interrupt
Processes and Kernel

Process 1
- Text
- Data
- Stack

Process 2
- Text
- Data
- Stack

Process n
- Text
- Data
- Stack

Kernel
- System Calls
  - fork
  - read
  - write

- System Management
  - Context Switching
  - Scheduling
Kernel Maintains Process Table

List of processes and their states

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Ready</th>
<th>(+ other state info)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 2</td>
<td>Running</td>
<td>...</td>
</tr>
<tr>
<td>Process 3</td>
<td>Ready</td>
<td>...</td>
</tr>
<tr>
<td>Process 4</td>
<td>Blocked</td>
<td>...</td>
</tr>
</tbody>
</table>

Other state info includes

- contents of CPU context
- areas of memory being used
- other information
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()
  • To block, call 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
How a Context Switch Occurs

Process makes system call (TRAP) or is interrupted
  • These are the only ways of entering the kernel

Then, via hardware
  • User-to-kernel mode switch: amplifies power
  • Go to fixed kernel location: interrupt/trap handler

Then, via software (in the kernel)
  • Save context of last-running process
  • Select new process from those that are ready
  • Restore context of selected process
  • RTI: return from interrupt/trap
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
System Management
Kernel-Level Threads

Process 1

Text

Data

Stack 1
Stack 2
Stack 3

System Calls

fork
read
write

Kernel

Process 2

Text

Data

Stack 1
Stack 2

Thread Context Switching
System Management

Process n

Text

Data

Stack 1

Thread + Process Scheduling

...