Goals for Today

• What are **processes**?
  – Process states and scheduling
  – Process creation
  – Process termination
  – Inter-process communication

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides courtesy of Prashant Shenoy, Anthony D. Joseph, John Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner.
Goal for Today

• What are threads?
• Classifying threaded systems
• Thread creation and finishing

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Slides courtesy of Anthony D. Joseph, John Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner.
Question

• Process is an instance of a program executing.
  – The fundamental OS responsibility

• Processes do their work by processing and calling file system operations

• Are their any operations on processes themselves?

• exit?
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024

int main(int argc, char *argv[]) {
    int c;

    pid_t pid = getpid(); /* get current processes PID */
    printf("My pid: %d\n", pid);

    c = fgetc(stdin);
    exit(0);
}
```
Can a process create a process?

• Yes
• Fork creates a copy of process
fork1.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024

int main(int argc, char *argv[])
{
    char buf[BUFSIZE];
    size_t readlen, writelen, slen;
    pid_t cpid, mypid;
    pid_t pid = getpid();         /* get current processes PID */
    printf("Parent pid: %d\n", pid);
    cpid = fork();
    if (cpid > 0) { /* Parent Process */
        mypid = getpid();
        printf("[%d] parent of [%d]\n", mypid, cpid);
    } else if (cpid == 0) { /* Child Process */
        mypid = getpid();
        printf("[%d] child\n", mypid);
    } else {
        perror("Fork failed");
        exit(1);
    }
    exit(0);
}
UNIX Process Management

• UNIX fork – system call to create a copy of the current process, and start it running
  – No arguments!
• UNIX exec – system call to change the program being run by the current process
• UNIX wait – system call to wait for a process to finish
• UNIX signal – system call to send a notification to another process
fork2.c

```c
...
cpid = fork();
if (cpid > 0) {   /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    tcpid = wait(&status);
    printf("[%d] bye %d\n", mypid, tcpid);
} else if (cpid == 0) {   /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
}
...```
UNIX Process Management

```c
main () {
...
}

fork

pid = fork();
if (pid == 0)
exec(...);
else
wait(pid);

wait

pid = fork();
if (pid == 0)
exec(...);
else
wait(pid);

exec

pid = fork();
if (pid == 0)
exec(...);
else
wait(pid);
```
Signals – infloop.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>

#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum) {
    printf("Caught signal %d – phew!\n", signum);
    exit(1);
}

int main() {
    signal(SIGINT, signal_callback_handler);

    while (1) {}
}
```
Process races: fork.c

```c
if (cpid > 0) {
    mypid = getpid();
    printf("[\%d] parent of [\%d]\n", mypid, cpid);
    for (i=0; i<100; i++) {
        printf("[\%d] parent: %d\n", mypid, i);
        //        sleep(1);
    }
} else if (cpid == 0) {
    mypid = getpid();
    printf("[\%d] child\n", mypid);
    for (i=0; i>-100; i--) {
        printf("[\%d] child: %d\n", mypid, i);
        //        sleep(1);
    }
}
```
Traditional UNIX Process

• Process: *Operating system abstraction to represent what is needed to run a single program*
  – Often called a “HeavyWeight Process”

• Two parts:
  – Sequential Program Execution Stream
    » Code executed as a *single, sequential* stream of execution (i.e., thread)
    » Includes State of CPU registers
  – Protected Resources:
    » Main Memory State (contents of Address Space)
    » I/O state (i.e. file descriptors)
How do we Multiplex Processes?

• The current state of process held in a process control block (PCB):
  – This is a “snapshot” of the execution and protection environment
  – Only one PCB active at a time

• Give out CPU time to different processes (Scheduling):
  – Only one process “running” at a time
  – Give more time to important processes

• Give pieces of resources to different processes (Protection):
  – Controlled access to non-CPU resources
  – Sample mechanisms:
    » Memory Mapping: Give each process their own address space
    » Kernel/User duality: Arbitrary multiplexing of I/O through system calls
CPU Switch From Process to Process

- This is also called a “context switch”
- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
As a process executes, it changes state

- **new**: The process is being created
- **ready**: The process is waiting to run
- **running**: Instructions are being executed
- **waiting**: Process waiting for some event to occur
- **terminated**: The process has finished execution
Process Scheduling

- PCBs move from queue to queue as they change state
  - Decisions about which order to remove from queues are **Scheduling** decisions
  - Many algorithms possible (few weeks from now)
What does it take to create a process?

• Must construct new PCB
  – Inexpensive

• Must set up new page tables for address space
  – More expensive

• Copy data from parent process? (Unix `fork()`)
  – Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state

• Copy I/O state (file handles, etc)
  – Medium expense
Modern “Lightweight” Process with Threads

• Thread: *a sequential execution stream within process* (Sometimes called a “Lightweight process”)
  – Process still contains a single Address Space
  – No protection between threads

• Multithreading: *a single program made up of a number of different concurrent activities*
  – Sometimes called multitasking, as in Ada …

• Why separate the concept of a thread from that of a process?
  – Discuss the “thread” part of a process (concurrency)
  – Separate from the “address space” (protection)
  – Heavyweight Process ≡ Process with one thread
Single and Multithreaded Processes

- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?
A typical use case

Client Browser
- process for each tab
- thread to render page
- GET in separate thread
- multiple outstanding GETs
- as they complete, render portion

Web Server
- fork process for each client connection
- thread to get request and issue response
- fork threads to read data, access DB, etc
- join and respond
Kernel Use Cases

• Thread for each user process
• Thread for sequence of steps in processing I/O
• Threads for device drivers
• …
Examples of multithreaded programs

• Embedded systems
  – Elevators, Planes, Medical systems, Wristwatches
  – Single Program, concurrent operations

• Most modern OS kernels
  – Internally concurrent because have to deal with concurrent requests by multiple users
  – But no protection needed within kernel

• Database Servers
  – Access to shared data by many concurrent users
  – Also background utility processing must be done
Examples of multithreaded programs (con’t)

• Network Servers
  – Concurrent requests from network
  – Again, single program, multiple concurrent operations
  – File server, Web server, and airline reservation systems

• Parallel Programming (More than one physical CPU)
  – Split program into multiple threads for parallelism
  – This is called Multiprocessing

• Some multiprocessors are actually uniprogrammed:
  – Multiple threads in one address space but one program at a time
Putting it together: Process

(Unix) Process

Sequential stream of instructions

A(int tmp) {
    if (tmp<2)
        B();
        printf(tmp);
    }
B() {
    C();
}
C() {
    A(2);
}
A(1);
...

CPU state (PC, SP, registers..)

Memory

Stack

I/O State (e.g., file, socket contexts)

Resources

Stored in OS
Putting it together: Processes

- Switch overhead: high
  - CPU state: low
  - Memory/IO state: high
- Process creation: high
- Protection
  - CPU: yes
  - Memory/IO: yes
- Sharing overhead: high
  (involves at least a context switch)
Putting it together: Threads

- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low)
Putting it together: Multi-Cores

- Switch overhead: low (only CPU state)
- Thread creation: low
- Protection
  - CPU: yes
  - Memory/IO: No
- Sharing overhead: low (thread switch overhead low)
Putting it together: Hyper-Threading

- Switch overhead between hardware-threads: very-low (done in hardware)
- Contention for ALUs/FPUs may hurt performance

8 threads at a time

© 2014 UCB CS377: Operating Systems Lec 2.29
Memory Footprint of Two-Thread Example

• If we stopped this program and examined it with a debugger, we would see
  – Two sets of CPU registers
  – Two sets of Stacks

• Questions:
  – Why not have a single stack for all threads?
  – How do we position stacks relative to each other?
  – What maximum size should we choose for the stacks?
  – How might you catch violations?
Thread Operations

• thread_fork(func, args)
  – Create a new thread to run func(args)
• thread_yield()
  – Relinquish processor voluntarily
• thread_join(thread)
  – In parent, wait for forked thread to exit, then return
• thread_exit
  – Quit thread and clean up, wake up joiner if any
Thread Abstraction

- Infinite number of processors
- Threads execute with variable speed
  - Programs must be designed to work with any schedule

Diagram:

Programmer Abstraction

<table>
<thead>
<tr>
<th>Threads</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Physical Reality

<table>
<thead>
<tr>
<th>Running Threads</th>
<th>Ready Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
## Programmer vs. Processor View

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution #1</th>
<th>Possible Execution #2</th>
<th>Possible Execution #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
<td>x = x + 1</td>
<td>x = x + 1</td>
</tr>
<tr>
<td>y = y + x;</td>
<td>y = y + x;</td>
<td>&quot;..............&quot;</td>
<td>y = y + x</td>
</tr>
<tr>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
<td>&quot;thread is suspended&quot;</td>
<td>&quot;..............&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;other thread(s) run&quot;</td>
<td>&quot;thread is resumed&quot;</td>
<td>&quot;other thread(s) run&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;thread is resumed&quot;</td>
<td></td>
<td>&quot;thread is resumed&quot;</td>
</tr>
</tbody>
</table>

y = y + x
z = x + 5y
Possible Executions

Thread 1 \[\square\] Thread 1 \[\square\]
Thread 2 \[\square\] Thread 2 \[\square\]
Thread 3 \[\square\] Thread 3 \[\square\]

a) One execution

Thread 1 \[\square\] \[\square\] \[\square\] \[\square\]
Thread 2 \[\square\] \[\square\] \[\square\] \[\square\]
Thread 3 \[\square\] \[\square\] \[\square\] \[\square\]

b) Another execution

c) Another execution
Thread State

• State shared by all threads in process/addr space
  – Content of memory (global variables, heap)
  – I/O state (file system, network connections, etc)

• State “private” to each thread
  – Kept in TCB ≡ Thread Control Block
  – CPU registers (including, program counter)
  – Execution stack – what is this?

• Execution Stack
  – Parameters, temporary variables
  – Return PCs are kept while called procedures are executing
Thread Lifecycle

- **Init**
  - Thread Creation
    - e.g., `sthread_create()`

- **Ready**
  - Event Occurs
    - e.g., other thread calls `sthread_join()`
  - Scheduler Resumes Thread
    - e.g., `sthread_yield()`

- **Running**
  - Scheduler Suspends Thread
    - e.g., `sthread_yield()`
  - Thread Waits for Event
    - e.g., `sthread_join()`

- **Finished**
  - Thread Exit
    - e.g., `sthread_exit()`
### Shared vs. Per-Thread State

<table>
<thead>
<tr>
<th>Shared State</th>
<th>Per–Thread State</th>
<th>Per–Thread State</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Heap</strong></td>
<td>Thread Control Block (TCB)</td>
<td>Thread Control Block (TCB)</td>
</tr>
<tr>
<td><strong>Global Variables</strong></td>
<td>Stack Information</td>
<td>Stack Information</td>
</tr>
<tr>
<td><strong>Code</strong></td>
<td>Saved Registers</td>
<td>Saved Registers</td>
</tr>
<tr>
<td></td>
<td>Thread Metadata</td>
<td>Thread Metadata</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td>Stack</td>
</tr>
</tbody>
</table>

© 2014 UCB

CS377: Operating Systems

Lec 2.37
Per Thread State

• Each Thread has a *Thread Control Block* (TCB)
  – Execution State: CPU registers, program counter (PC), pointer to stack (SP)
  – Scheduling info: state, priority, CPU time
  – Various Pointers (for implementing scheduling queues)
  – Pointer to enclosing process (PCB)
  – Etc (add stuff as you find a need)

• OS Keeps track of TCBs in protected memory
  – In Array, or Linked List, or …
Multithreaded Processes

- PCB points to multiple TCBs:

- Switching threads within a block is a simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables.
The Numbers

Context switch in Linux: 3-4 μsecs (Current Intel i7 & E5).

• Thread switching faster than process switching (100 ns).
• But switching across cores about 2x more expensive than within-core switching.
• Context switch time increases sharply with the size of the working set*, and can increase 100x or more.

* The working set is the subset of memory used by the process in a time window.

Moral: Context switching depends mostly on cache limits and the process or thread’s hunger for memory.
### Classification

<table>
<thead>
<tr>
<th># threads Per AS:</th>
<th># of addr spaces:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
<td></td>
</tr>
<tr>
<td>Many</td>
<td>Embedded systems (Geoworks, VxWorks, JavaOS, etc) JavaOS, Pilot(PC)</td>
<td>Mach, OS/2, HP-UX, Win NT to 8, Solaris, OS X, Android, iOS</td>
<td></td>
</tr>
</tbody>
</table>

- Real operating systems have either
  - One or many address spaces
  - One or many threads per address space
Dispatch Loop

• Conceptually, the dispatching loop of the operating system looks as follows (loop runs in kernel mode if kernel thread and user mode if user thread):

```c
Loop {
    RunThread();
    SaveStateOfCPU(curTCB);
    run_new_thread();
    LoadStateOfCPU(newTCB);
}
```

• This is an *infinite* loop
  – One could argue that this is all that the OS does
Running a thread

Consider first portion: \texttt{RunThread()}

- How do I run a thread?
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC

- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets \textit{preempted}
Creating a new thread

/* thread_create: user-level procedure that creates a new thread and places it on the ready queue */
thread_create(pointer_to_procedure, arg0, ...){

    /* Allocate a new TCB and execution call stack */
    TCB tcb = new TCB();
    Stack stack = new Stack();

    /* Initialize TCB and stack with initial register values and the address of the first instruction to run */
    tcb.PC = stub;
    tcb.stack = stack;
    tcb.arg0Reg = procedure; /* ptr to application routine */
    tcb.arg1Reg = arg0;        /* arguments for the routine */
    tcb.arg2Reg = arg1;
    ..

    /* Put thread on the ready list */
    readyQ.add(tcb);
}
What does `ThreadRoot stub()` look like?

```c
stub(proc, arg0, arg1, ...){
    (*proc)(arg0, arg1, ...);
    deleteCurrentThread();
}
```

[Diagram of thread creation and stub function]
What does `deleteCurrentThread()` do?

- Needs to re-enter kernel mode (system call)
- “Wake up” (place on ready queue) threads waiting for this thread (e.g. parent waiting for thread to finish)
- Can’t deallocate thread yet
  - We are still running on its stack!
  - Instead, record thread as “waitingToBeDestroyed”
- Call `run_new_thread()` to run another thread:
  ```
  run_new_thread() {
      newThread = PickNewThread();
      switch(curThread, newThread);
      ThreadHouseKeeping();
  }
  ```
  - `ThreadHouseKeeping()` notices `waitingToBeDestroyed` and deallocates the finished thread’s TCB and stack
ThreadJoin() system call

- One thread can wait for another to finish with the ThreadJoin(tid) call
  - Calling thread will be taken off run queue and placed on waiting queue for thread tid
- Where is a logical place to store this wait queue?
  - On queue inside the TCB

- Similar to wait() system call in UNIX
  - Lets parents wait for child processes
Use of Join for Traditional Procedure Call

• A traditional procedure call is logically equivalent to doing a ThreadFork followed by ThreadJoin

• Consider the following normal procedure call of B() by A():

```pseudocode
A() { B(); }
B() { Do interesting, complex stuff }
```

• The procedure A() is equivalent to A’():

```pseudocode
A’() {
    tid = ThreadFork(B,null);
    ThreadJoin(tid);
}
```

• Why not do this for every procedure?
  – Context Switch Overhead
  – Memory Overhead for Stacks
Summary

• Multithreading provides simple and lightweight illusion of multiple CPUs
• Threads may be at user-level or kernel level
• New Threads Created with `ThreadFork stub()`
  – Create initial TCB and stack to point at `ThreadRoot()`
  – `ThreadRoot()` calls thread code, then `deleteCurrentThread()`
  – `deleteCurrentThread()` wakes up waiting threads then prepares TCB/stack for destruction
• Threads can wait for other threads using `ThreadJoin()`