

CSI62
Operating Systems and
Systems Programming
Lecture 5

Introduction to Networking,
Concurrency (Processes and Threads)

September 11th, 2017
Prof. Ion Stoica
<http://cs162.eecs.berkeley.edu>

So what happens when you fgetc?

Application / Service

High Level I/O

Low Level I/O

Syscall

File System

I/O Driver

streams
handles
registers

descriptors

Commands and Data Transfers

Disks, Flash, Controllers, DMA



9/11/17

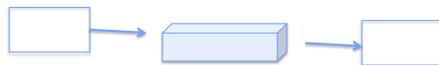
CS162 ©UCB Fall 2017

Lec 5.2

Communication between processes

- Can we view files as communication channels?

```
write(wfd, wbuf, wlen);
```



```
n = read(rfd, rbuf, rmax);
```

- Producer and Consumer of a file may be distinct processes
 - May be separated in time (or not)
- However, what if data written once and consumed once?
 - Don't we want something more like a queue?
 - Can still look like File I/O!

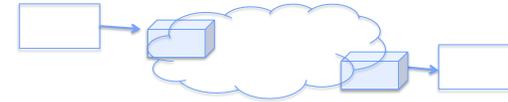
9/11/17

CS162 ©UCB Fall 2017

Lec 5.3

Communication Across the world looks like file I/O

```
write(wfd, wbuf, wlen);
```



```
n = read(rfd, rbuf, rmax);
```

- Connected queues over the Internet
 - But what's the analog of open?
 - What is the namespace?
 - How are they connected in time?

9/11/17

CS162 ©UCB Fall 2017

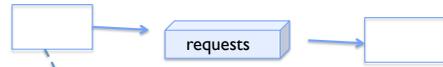
Lec 5.4

Request Response Protocol

Client (issues requests)

Server (performs operations)

```
write(rqfd, rqbuf, buflen);
```



```
n = read(rfd, rbuf, rmax);
```

service request

```
write(wfd, respbuf, len);
```



```
n = read(resfd, resbuf, resmax);
```

9/11/17

CS162 @UCB Fall 2017

Lec 5.5

Request Response Protocol

Client (issues requests)

Server (performs operations)

```
write(rqfd, rqbuf, buflen);
```



```
n = read(rfd, rbuf, rmax);
```

service request

```
write(wfd, respbuf, len);
```



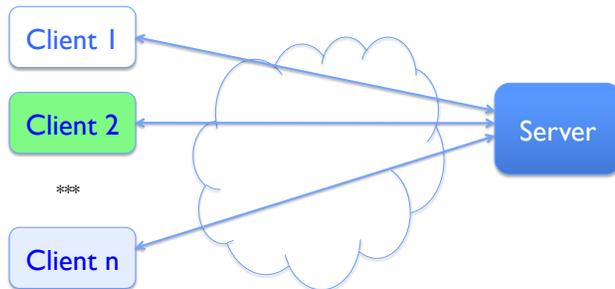
```
n = read(resfd, resbuf, resmax);
```

9/11/17

CS162 @UCB Fall 2017

Lec 5.6

Client-Server Models



- File servers, web, FTP, Databases, ...
- Many clients accessing a common server

9/11/17

CS162 @UCB Fall 2017

Lec 5.7

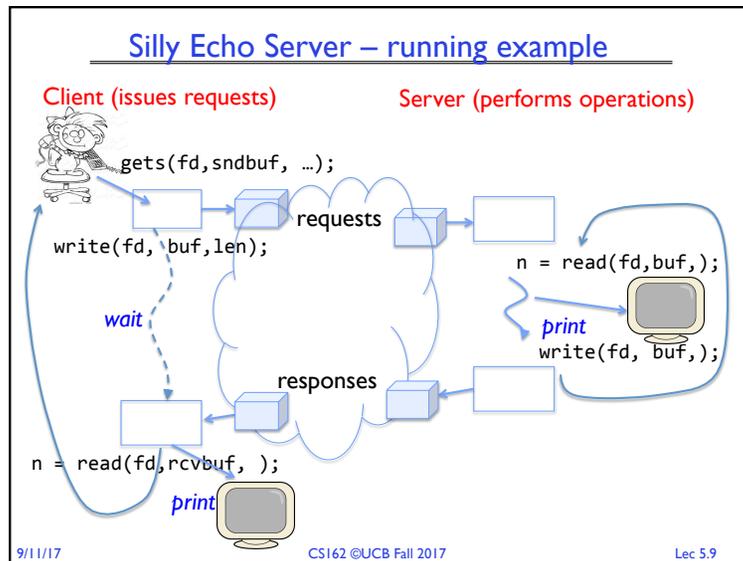
Sockets

- **Socket:** an abstraction of a network I/O queue
 - Mechanism for inter-process communication
 - Embodies one side of a communication channel
 - » Same interface regardless of location of other end
 - » Local machine ("UNIX socket") or remote machine ("network socket")
 - First introduced in 4.2 BSD UNIX: big innovation at time
 - » Now most operating systems provide some notion of socket
- Data transfer like files
 - Read / Write against a descriptor
- Over ANY kind of network
 - Local to a machine
 - Over the internet (TCP/IP, UDP/IP)
 - OSI, Appletalk, SNA, IPX, SIP, NS, ...

9/11/17

CS162 @UCB Fall 2017

Lec 5.8



Echo client-server example

```

void client(int sockfd) {
  int n;
  char sndbuf[MAXIN]; char rcvbuf[MAXOUT];
  getreq(sndbuf, MAXIN); /* prompt */
  while (strlen(sndbuf) > 0) {
    write(sockfd, sndbuf, strlen(sndbuf)); /* send */
    memset(rcvbuf, 0, MAXOUT); /* clear */
    n = read(sockfd, rcvbuf, MAXOUT-1); /* receive */
    write(STDOUT_FILENO, rcvbuf, n); /* echo */
    getreq(sndbuf, MAXIN); /* prompt */
  }
}

void server(int consockfd) {
  char reqbuf[MAXREQ];
  int n;
  while (1) {
    memset(reqbuf, 0, MAXREQ);
    n = read(consockfd, reqbuf, MAXREQ-1); /* Recv */
    if (n <= 0) return;
    n = write(STDOUT_FILENO, reqbuf, strlen(reqbuf));
    n = write(consockfd, reqbuf, strlen(reqbuf)); /* echo */
  }
}
  
```

9/11/17 CS162 @UCB Fall 2017 Lec 5.10

Prompt for input

```

char *getreq(char *inbuf, int len) {
  /* Get request char stream */
  printf("REQ: "); /* prompt */
  memset(inbuf, 0, len); /* clear for good measure */
  return fgets(inbuf, len, stdin); /* read up to a EOL */
}
  
```

9/11/17 CS162 @UCB Fall 2017 Lec 5.11

- ### Socket creation and connection
- File systems provide a collection of permanent objects in structured name space
 - Processes open, read/write/close them
 - Files exist independent of the processes
 - Sockets provide a means for processes to communicate (transfer data) to other processes.
 - Creation and connection is more complex
 - Form 2-way pipes between processes
 - Possibly worlds away
- 9/11/17 CS162 @UCB Fall 2017 Lec 5.12

Namespaces for communication over IP

- Hostname
 - www.eecs.berkeley.edu
- IP address
 - 128.32.244.172 (IPv4 32-bit)
 - fe80::4ad7:5ff:febf:2607 (IPv6 128-bit)
- Port Number
 - 0-1023 are “well known” or “system” ports
 - » Superuser privileges to bind to one
 - 1024 – 49151 are “registered” ports ([registry](#))
 - » Assigned by IANA for specific services
 - 49152–65535 ($2^{15}+2^{14}$ to $2^{16}-1$) are “dynamic” or “private”
 - » Automatically allocated as “ephemeral Ports”

9/11/17

CS162 @UCB Fall 2017

Lec 5.13

Using Sockets for Client-Server (C/C++)

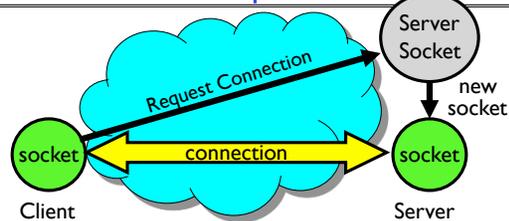
- On server: set up “server-socket”
 - Create socket; bind to protocol (TCP), local address, port
 - Call **listen()**: tells server socket to accept incoming requests
 - Perform multiple **accept()** calls on socket to accept incoming connection request
 - Each successful **accept()** returns a new socket for a new connection; can pass this off to handler thread
- On client:
 - Create socket; bind to protocol (TCP), remote address, port
 - Perform **connect()** on socket to make connection
 - If **connect()** successful, have socket connected to server

9/11/17

CS162 @UCB Fall 2017

Lec 5.14

Socket Setup over TCP/IP



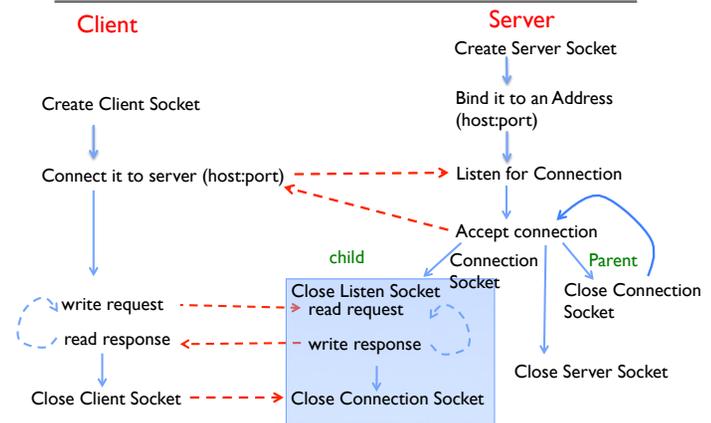
- Server Socket: Listens for new connections
 - Produces new sockets for each unique connection
- Things to remember:
 - Connection involves 5 values: [Client Addr, Client Port, Server Addr, Server Port, Protocol]
 - Often, Client Port “randomly” assigned by OS during client socket setup
 - Server Port often “well known” (0-1023)
 - » 80 (web), 443 (secure web), 25 (sendmail), etc

9/11/17

CS162 @UCB Fall 2017

Lec 5.15

Example: Server Protection and Parallelism



9/11/17

CS162 @UCB Fall 2017

Lec 5.16

Server Protocol (v3)

```
listen(lstnsockfd, MAXQUEUE);
while (1) {
    consockfd = accept(lstnsockfd, (struct sockaddr *) &cli_addr,
                       &cli_len);

    cpid = fork();           /* new process for connection */
    if (cpid > 0) {         /* parent process */
        close(consockfd);
        //tcpid = wait(&cstatus);
    } else if (cpid == 0) { /* child process */
        close(lstnsockfd); /* let go of listen socket */

        server(consockfd);

        close(consockfd);
        exit(EXIT_SUCCESS); /* exit child normally */
    }
}
close(lstnsockfd);
```

9/11/17

CS162 @UCB Fall 2017

Lec 5.17

Server Address - Itself

```
struct sockaddr_in {
    short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
} serv_addr;

memset((char *) &serv_addr, 0, sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = INADDR_ANY;
serv_addr.sin_port = htons(portno);
```

- Simple form
- Internet Protocol
- accepting any connections on the specified port
- In “network byte ordering” (which is *big endian*)

9/11/17

CS162 @UCB Fall 2017

Lec 5.18

Client: Getting the Server Address

```
struct hostent *buildServerAddr(struct sockaddr_in *serv_addr,
                               char *hostname, int portno) {
    struct hostent *server;

    /* Get host entry associated with a hostname or IP address */
    server = gethostbyname(hostname);
    if (server == NULL) {
        fprintf(stderr, "ERROR, no such host\n");
        exit(1);
    }

    /* Construct an address for remote server */
    memset((char *) serv_addr, 0, sizeof(struct sockaddr_in));
    serv_addr->sin_family = AF_INET;
    bcopy((char *)server->h_addr,
          (char *)&(serv_addr->sin_addr.s_addr), server->h_length);
    serv_addr->sin_port = htons(portno);

    return server;
}
```

9/11/17

CS162 @UCB Fall 2017

Lec 5.19

Administrivia

- **TA preferences due tonight at 11:59PM**
 - We will try to accommodate your needs, but have to balance both over-popular and under-popular sections
- Attend section and get to know your TAs!

9/11/17

CS162 @UCB Fall 2017

Lec 5.20

BREAK

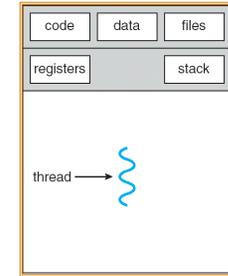
9/11/17

CS162 @UCB Fall 2017

Lec 5.21

Recall: Traditional UNIX Process

- Process: OS abstraction of what is needed to run a single program
 - Often called a “Heavyweight Process”
 - No concurrency in a “Heavyweight Process”
- Two parts:
 - Sequential program execution stream [ACTIVE PART]
 - » Code executed as a sequential stream of execution (i.e., thread)
 - » Includes State of CPU registers
 - Protected resources [PASSIVE PART]:
 - » Main memory state (contents of Address Space)
 - » I/O state (i.e. file descriptors)



9/11/17

CS162 @UCB Fall 2017

Lec 5.22

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
 - Example mechanisms:
 - » Memory Mapping: Give each process their own address space
 - » Kernel/User duality: Arbitrary multiplexing of I/O through system calls



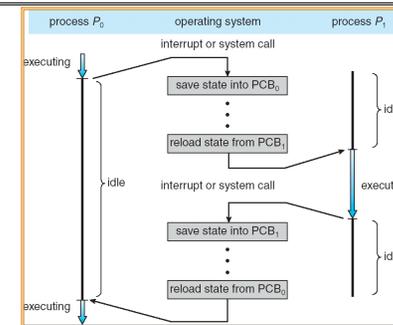
Process Control Block

9/11/17

CS162 @UCB Fall 2017

Lec 5.23

CPU Switch From Process A to Process B



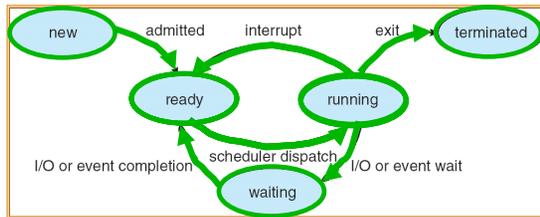
- This is also called a “context switch”
- Code executed in kernel above is **overhead**
 - Overhead sets minimum practical switching time
 - Less overhead with SMT/hyperthreading, but... contention for resources instead

9/11/17

CS162 @UCB Fall 2017

Lec 5.24

Lifecycle of a Process



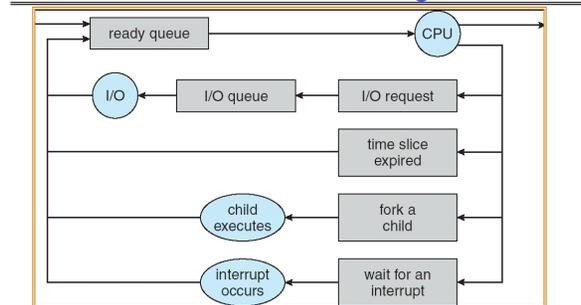
- As a process executes, it changes state:
 - **new**: The process is being created
 - **ready**: The process is waiting to run
 - **running**: Instructions are being executed
 - **waiting**: Process waiting for some event to occur
 - **terminated**: The process has finished execution

9/11/17

CS162 @UCB Fall 2017

Lec 5.25

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)

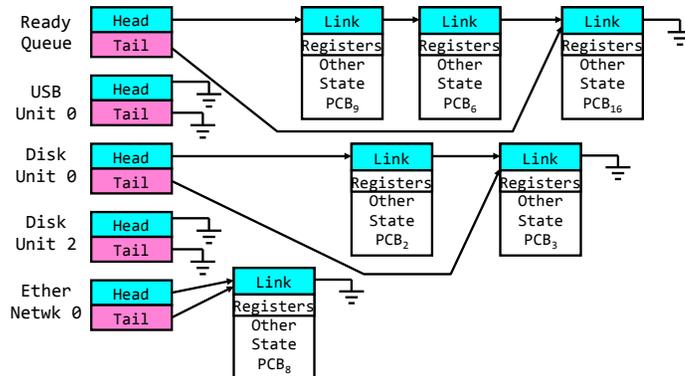
9/11/17

CS162 @UCB Fall 2017

Lec 5.26

Ready Queue And Various I/O Device Queues

- Process not running \Rightarrow PCB is in some scheduler queue
 - Separate queue for each device/signal/condition
 - Each queue can have a different scheduler policy



9/11/17

CS162 @UCB Fall 2017

Lec 5.27

Modern Process with Threads

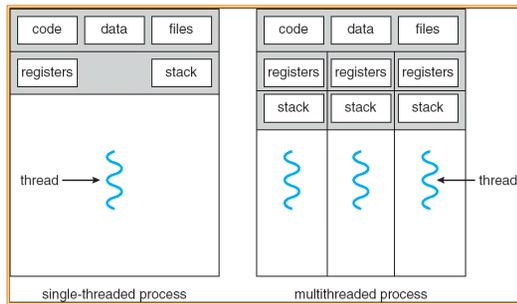
- Thread: *a sequential execution stream within process* (Sometimes called a "**Lightweight process**")
 - Process still contains a single Address Space
 - No protection between threads
- Multithreading: *a single program made up of a number of different concurrent activities*
 - Sometimes called multitasking, 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

9/11/17

CS162 @UCB Fall 2017

Lec 5.28

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?

9/11/17

CS162 @UCB Fall 2017

Lec 5.29

Thread State

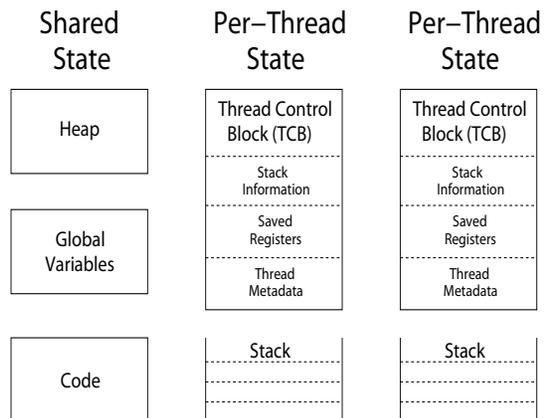
- State shared by all threads in process/address space
 - Content of memory (global variables, heap)
 - I/O state (file descriptors, network connections, etc)
- State “private” to each thread
 - Kept in **TCB = Thread Control Block**
 - CPU registers (including, program counter)
 - Execution stack – what is this?
- Execution Stack
 - Parameters, temporary variables
 - Return PCs are kept while called procedures are executing

9/11/17

CS162 @UCB Fall 2017

Lec 5.30

Shared vs. Per-Thread State

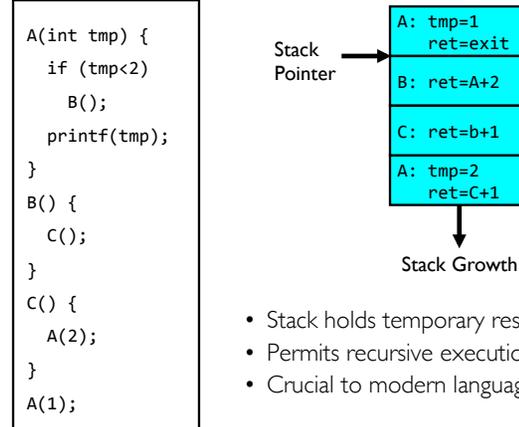


9/11/17

CS162 @UCB Fall 2017

Lec 5.31

Execution Stack Example



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

9/11/17

CS162 @UCB Fall 2017

Lec 5.32

MIPS: Software conventions for Registers

0	zero	constant 0	16	s0	callee saves
1	at	reserved for assembler	(callee must save)
2	v0	expression evaluation &	23	s7	
3	v1	function results	24	t8	temporary (cont'd)
4	a0	arguments	25	t9	
5	a1		26	k0	reserved for OS kernel
6	a2		27	k1	
7	a3		28	gp	Pointer to global area
8	t0	temporary: caller saves	29	sp	Stack pointer
...	...	(callee can clobber)	30	fp	frame pointer
15	t7		31	ra	Return Address (HW)

- Before calling procedure:
 - Save caller-saves regs
 - Save v0, v1
 - Save ra
- After return, assume
 - Callee-saves reg OK
 - gp,sp,fp OK (restored!)
 - Other things trashed

9/11/17

CS162 @UCB Fall 2017

Lec 5.33

Motivational Example for Threads

- Imagine the following C program:

```
main() {
    ComputePI("pi.txt");
    PrintClassList("classlist.txt");
}
```

- What is the behavior here?
 - Program would never print out class list
 - Why? **ComputePI** would never finish

9/11/17

CS162 @UCB Fall 2017

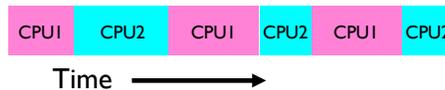
Lec 5.34

Use of Threads

- Version of program with Threads (loose syntax):

```
main() {
    ThreadFork(ComputePI, "pi.txt");
    ThreadFork(PrintClassList, "classlist.txt");
}
```

- What does **ThreadFork()** do?
 - Start independent thread running given procedure
- What is the behavior here?
 - Now, you would actually see the class list
 - This *should* behave as if there are two separate CPUs



9/11/17

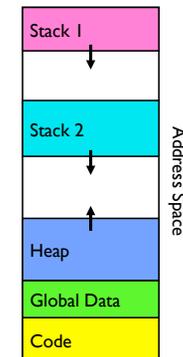
CS162 @UCB Fall 2017

Lec 5.35

Memory Footprint: Two-Threads

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

- Questions:
 - How do we position stacks relative to each other?
 - What maximum size should we choose for the stacks?
 - What happens if threads violate this?
 - How might you catch violations?



9/11/17

CS162 @UCB Fall 2017

Lec 5.36

Actual Thread Operations

- **thread_fork(func, args)**
 - Create a new thread to run func(args)
 - Pintos: **thread_create**
- **thread_yield()**
 - Relinquish processor voluntarily
 - Pintos: **thread_yield**
- **thread_join(thread)**
 - In parent, wait for forked thread to exit, then return
 - Pintos: **thread_join**
- **thread_exit**
 - Quit thread and clean up, wake up joiner if any
 - Pintos: **thread_exit**
- **pThreads**: POSIX standard for thread programming [POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)]

9/11/17

CS162 @UCB Fall 2017

Lec 5.37

Dispatch Loop

- Conceptually, the dispatching loop of the operating system looks as follows:

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

- This is an *infinite* loop
 - One could argue that this is all that the OS does
- Should we ever exit this loop???
 - When would that be?

9/11/17

CS162 @UCB Fall 2017

Lec 5.38

Running a thread

Consider first portion: **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 *preempted*

9/11/17

CS162 @UCB Fall 2017

Lec 5.39

Internal Events

- Blocking on I/O
 - The act of requesting I/O implicitly yields the CPU
- Waiting on a “signal” from other thread
 - Thread asks to wait and thus yields the CPU
- Thread executes a **yield()**
 - Thread volunteers to give up CPU

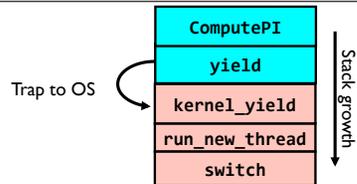
```
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```

9/11/17

CS162 @UCB Fall 2017

Lec 5.40

Stack for Yielding Thread



- How do we run a new thread?


```
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Do any cleanup */
}
```
- How does dispatcher switch to a new thread?
 - Save anything next thread may trash: PC, regs, stack pointer
 - Maintain isolation for each thread

9/11/17

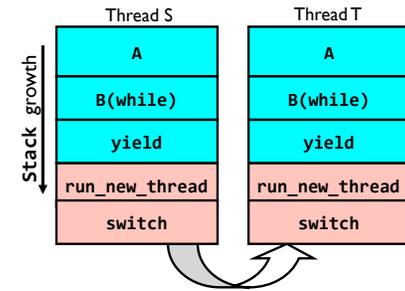
CS162 @UCB Fall 2017

Lec 5.41

What Do the Stacks Look Like?

- Consider the following code blocks:

```
proc A() {
    B();
}
proc B() {
    while(TRUE) {
        yield();
    }
}
```



- Suppose we have 2 threads:
 - Threads S and T

9/11/17

CS162 @UCB Fall 2017

Lec 5.42

Saving/Restoring state (often called "Context Switch")

```
Switch(tCur, tNew) {
    /* Unload old thread */
    TCB[tCur].regs.r7 = CPU.r7;
    ...
    TCB[tCur].regs.r0 = CPU.r0;
    TCB[tCur].regs.sp = CPU.sp;
    TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

    /* Load and execute new thread */
    CPU.r7 = TCB[tNew].regs.r7;
    ...
    CPU.r0 = TCB[tNew].regs.r0;
    CPU.sp = TCB[tNew].regs.sp;
    CPU.retpc = TCB[tNew].regs.retpc;
    return; /* Return to CPU.retpc */
}
```

9/11/17

CS162 @UCB Fall 2017

Lec 5.43

Switch Details (continued)

- What if you make a mistake in implementing switch?
 - Suppose you forget to save/restore register 32
 - Get intermittent failures depending on when context switch occurred and whether new thread uses register 32
 - System will give wrong result without warning
- Can you devise an exhaustive test to test switch code?
 - No! Too many combinations and inter-leavings
- Cautionary tale:
 - For speed, Topaz kernel saved one instruction in switch()
 - Carefully documented! Only works as long as kernel size < 1MB
 - What happened?
 - Time passed, People forgot
 - Later, they added features to kernel (no one removes features!)
 - Very weird behavior started happening
 - Moral of story: Design for simplicity

9/11/17

CS162 @UCB Fall 2017

Lec 5.44

Summary

- Processes have two parts
 - One or more Threads (Concurrency)
 - Address Spaces (Protection)
- Concurrency accomplished by multiplexing CPU Time:
 - Unloading current thread (PC, registers)
 - Loading new thread (PC, registers)
 - Such context switching may be voluntary (`yield()`, I/O operations) or involuntary (timer, other interrupts)