Computer Architecture and Organization

Introduction to Operating Systems

Henk Corporaal & Ben Juurlink
Delft University of Technology
March 2000

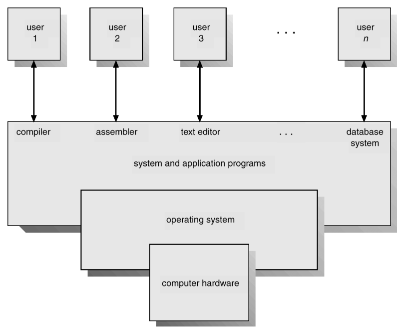
Objectives

- After this lecture, you should be able to
  - tell what an operating system is and what it does (subtitle of textbook Patterson and Hennessy is “Hardware/software interface”)
  - name and describe the most important OS components
  - write small C-programs that invoke Linux system calls
  - sketch how time sharing is implemented
  - recognize the synchronization problem associated with shared data and solve it using semaphores
  - understand how multithreading is implemented

What is an Operating System?

- Program that acts as intermediary between user of computer and computer hardware.
- Operating system goals:
  - Execute user programs and make solving problems easier.
  - Make the computer system convenient to use.
  - Use the computer hardware in an efficient manner.
- **Kernel** – the one program running at all times (all else being application programs).

Computer System



Layered View (Tanenbaum)

Problem-Oriented Language	Level 5
Assembly Language	Level 4
Operating System	Level 3
Instruction Set Architecture	Level 2
Microarchitecture	Level 1
Digital Logic	Level 0

System Calls

- **System call** – the method used by a process to request action by the OS
  - Control is given to the OS (**trap**)
  - OS services request
  - Control is returned to user program
- System calls provide the interface between a running program and the OS
  - Generally available as assembly-language instructions
  - In C, system calls can be made directly
- Typically, I/O is implemented through system calls, because
  - performing I/O directly is complex (many different devices)
  - protection is needed

Common OS Components

- **Process Management**
- **Memory Management**
- File Management
- I/O Management
- **Protection**
- Networking
- Command-Interpreter System

Process Management

- A *process* is a program in execution. A process needs certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
- OS is responsible for the following activities:
  - Process creation and deletion.
  - process suspension and resumption.
  - Provision of mechanisms for:
    - process synchronization
    - process communication
- Linux system calls: `fork, exec*, wait, exit, getpid, ...`

Memory Management

- OS is responsible for the following activities:
  - Keep track of which parts of memory are currently being used and by whom.
  - Decide which processes or parts of processes to load when memory space becomes available.
  - Allocate and deallocate memory space as needed.
- Linux system calls: `brk, mmap, munmap, ...`

File Management

- A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data
- OS is responsible for the following activities:
  - File creation and deletion.
  - Directory creation and deletion.
  - Support of primitives for manipulating files and directories.
  - Mapping files onto secondary storage.
- Linux system calls: `creat, open, close, mkdir, read, write, ...`

Protection

- *Protection* refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.
- The protection mechanism must:
  - distinguish between authorized and unauthorized usage.
  - specify the controls to be imposed.
  - provide a means of enforcement.
- In Linux, *file protection* is done using so-called rwx-bits for owner, group, world (, root)
- Linux system calls: `chmod, chown`

Networking (Distributed Systems)

- A *distributed* system is a collection processors that do not share memory or a clock. Each processor has its own local memory.
- The processors in the system are connected through a communication network.
- Access to a shared resource allows:
  - Computation speed-up
  - Increased data availability
  - Enhanced reliability
  - Communication
- Linux system calls: `socket, connect, listen, accept, read, write, ...`

Command-Interpreter System

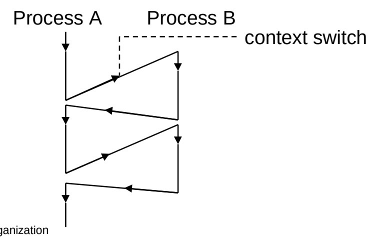
- It is possible to give commands to OS that deal with:
  - process creation and management (*command*, `ps, top, ...`)
  - file management (`cp, ls, cat, ...`)
  - protection (`chmod, chgrp`)
  - networking (`finger, mail, telnet, ...`)
- In Linux, the program that reads and interprets commands is called **shell**

Real-Time Operating Systems

- Often used as a control device in a dedicated application such as controlling scientific experiments, medical imaging systems, industrial control systems, ...
- Well-defined fixed-time constraints.
- *Hard real-time system*.
  - Secondary storage limited or absent, data stored in short-term memory, or read-only memory (ROM)
  - Conflicts with time-sharing systems, not supported by general-purpose operating systems.
- *Soft real-time system*
  - Limited utility in industrial control or robotics
  - Useful in applications (multimedia, virtual reality) requiring advanced operating-system features.

Concurrent Processing

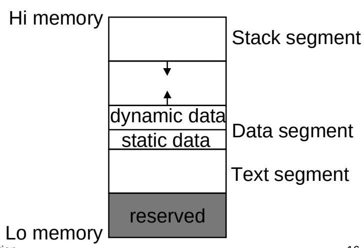
- Process = program in execution
- Modern operating systems allow many processes to be running at the same time
- Simulated concurrent processing / time sharing
  - Parallel processing simulated on one CPU



Processes and Virtual Memory

- Read sections 2.1 and 2.2 yourself
- Important: each process has its own (virtual) address space (there is a separate page table for each process)
- UNIX/Linux divides memory space into three parts:

Do you now understand why MIPS stack grows from low to high addresses?



Reasons for Supporting Time Sharing

- Overlapping I/O with computation
- Sharing CPU among several users
- Some programming problems can most naturally be represented as a collection of parallel processes
  - Web browser
  - Airline reservation system
  - Embedded µcontroller
  - Window manager
  - Garbage collection

An Example: Linux

Linux Online! http://www.linux.org

- When you give a command to the shell, a new process is started that executes the command
- Two options
  - Wait for the command to terminate
    `os_prompt> netscape`
  - Do not wait (run in background)
    `os_prompt> netscape&`
- One interprocess communication mechanism (IPC) is the **pipe**
  - Example:
    `os_prompt> ls | wc -w`

Process Creation in Linux

- Process can create **child process** by executing `fork()` system call
- **Parent process** does not wait for child, both of them run (pseudo-) concurrently
- Immediately after `fork`, child is *exact clone* of parent, i.e., text and data segments are identical
- Which is which?
  - Process IDentifier (PID)
  - `pid_t getpid(void);` returns PID of calling process
  - return value of `fork` is
    - child's PID for parent
    - 0 for child

Process Creation in Linux (cont.)

```
#include <unistd.h>
#include <sys/types.h>

main()
{
  pid_t pid_value;

  printf("PID before fork(): %d\n", (int)getpid());

  pid_value = fork();
  if (pid_value==0)
    printf("Hello from child. PID = %d\n", (int)getpid());
  else
    printf("Hello from parent. PID = %d\n", (int)getpid());
}
```


Process Creation in Linux (cont.)

- Shell forks a new process when you enter a command
- Child process is exact duplicate of the shell
- How do we get it to execute the command?
- System call

  `int execv(char *pathname, char **argv);`

  replaces text and data segments by some other program
  - `pathname` is the full path of the command
  - `argv` contains the command-line parameters

Process Creation in Linux (cont.)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

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

  if (argc==1){
    printf("Usage: run <command> [<parameters>]\n");
    exit(1);
  }

  pid_value = fork();
  if (pid_value==0){ /* Child */
    execv(argv[1], &argv[1]);
    printf("Sorry, couldn't run that\n");
  }
}
```


Process Creation in Linux (cont.)

- Example usage

```
os_prompt> run ls -l
Sorry, couldn't run that
os_prompt> run /bin/ls -l
total 1
-rw-r--r--  1 ben  other   64045 May   9 18:30 caos3.tex
```


Process Creation in Linux (cont.)

- Sometimes we want that parent waits for child to terminate
- System call

  `pid_t wait(int *status);`

  blocks calling process until one of its children terminates
- Return value is PID of child

Process Creation in Linux (cont.)

```
main(int argc, char *argv[])
{
  pid_t pid_value;
  int   status;

  if (argc==1){
    printf("Usage: run <command> [<parameters>]\n");
    exit(1);
  }

  pid_value = fork();
  if (pid_value==0){ /* Child */
    execv(argv[1], &argv[1]);
    printf("Sorry, couldn't run that\n");
  }
  else /* Parent */
    wait(&status);
}
```

computer architecture and organization 25

IPC in Linux

- Simple IPC mechanism: **pipe**
- Pipe is a fixed-size buffer which can be read/written like a file (i.e., sequentially / byte-for-byte)
- System calls:
  `int pipe(int fd[2]);`
  creates a new pipe
  - return value: -1 on error
  - `fd`: two file descriptors
    - `fd[0]`: read-end of the pipe
    - `fd[1]`: write-end of the pipe

  `int read(int fd, void *buf, size_t nbytes);`
  `int write(int fd, void *buf, size_t nbytes);`
- If process attempts to read from empty pipe or write to full pipe, it is blocked

computer architecture and organization 26

IPC in Linux (cont.)

```
main()
{
  int fd[2], i, status;
  char c, msg[13] = "Hello world!";

  if (pipe(fd)==-1){
    printf("Error creating pipe\n");
    exit(1);
  }

  if (fork()){ /* parent process is pipe writer */
    close(fd[0]); /* close read-end of pipe */
    for (i=0; i<12; i++) write(fd[1], &msg[i], 1);
    wait(&status);
  }
  else { /* child process is pipe reader */
    close(fd[1]);
    for (i=0; i<12; i++){
      read(fd[0], &c, 1); printf("%c", c);
    }
    printf("\n");
  }
}
```

computer architecture and organization 27

Implementation of Time Sharing

- Do we need special instructions?
- What about `jr $k0`?
- Processor state (register contents, PC, page table register,...) must be readable/writeable
- **Process control block (PCB)**: data structure in which OS stores context of a process
  - value of PC at the time process was interrupted
  - contents of the registers
  - page table register
  - ...
  - bookkeeping information
  - What about condition code register?

computer architecture and organization 28

Implementation of Time Sharing (cont.)

- Electronic timer (clock) periodically generates interrupt
- address of interrupted instruction is saved in `$epc`
- control is transferred to interrupt handler, which
  - saves the register contents in PCB
  - moves `$epc` to general purpose register (why?) and stores it in PCB
  - saves other context information in PCB
  - selects new process to run (CPU scheduling algorithm)
  - loads context of new process
  - flushes the TLB (why?)
  - loads saved `$epc` in `$k0` or `$k1` and transfers control to the new process by executing `jr $k0` or `jr $k1`

computer architecture and organization 29

Implementation of Time Sharing (cont.)

- Time for context switch can be large (1 to 1000 µsec.)
- Overhead is caused by
  - registers need to be saved and restored
    - DECSYSTEM-20: multiple register sets
  - TLB needs to be flushed
    - Add PID to each virtual address
    - TLB hit: both page number and PID match

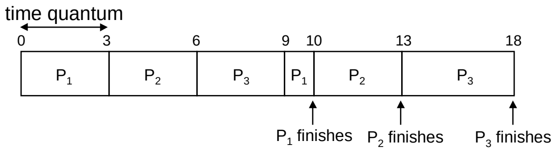
computer architecture and organization 30

CPU Scheduling

- Goals
  - Fairness
  - Efficiency
  - Maximize throughput
  - Minimize response time
- Round Robin: processes are given control of the CPU in a circular fashion. If a process uses up its *time quantum*, it is taken away from the CPU and put on the end of a list of processes

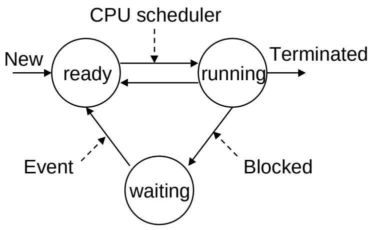
CPU Scheduling (cont.)

- Round robin example
  - P_1 takes 4 time units
  - P_2 takes 6 time units
  - P_3 takes 8 time units



Process Scheduling (cont.)

- To improve efficiency and throughput, context switch is also performed when a process is blocked (e.g., when it generated a page fault)



Protection

- If several user processes can be in memory simultaneously, OS must ensure that incorrect or malicious program cannot cause other programs to execute incorrectly
- Provide hardware support (**mode bit**) to differentiate at least two modes of operations
  - **User mode** - execution on behalf of user
  - **System mode** (also **kernel** or **supervisor** mode) - execution on behalf of OS
- **Privileged instructions** can only be executed in system mode

Protection (cont.)

- I/O instructions are privileged
- What about "memory protection"?
  - Systems with paging: process cannot access pages belonging to other processes (all memory accesses must go through page table)
- How to enforce?
  - Processes must be forbidden to change page table
  - OS must be able to modify page tables
- Solution
  - Place page tables in address space of OS
  - Make "load page table register" a privileged instruction

Process Synchronization

- Background
- The Critical Section Problem
- Synchronization Hardware
- Semaphores
- Classical Synchronization Problems

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure orderly execution of cooperating processes
- Linux processes cannot directly communicate via shared variables (why?). **Threads** (discussed later) can.

computer architecture and organization 37

Background (cont.)

- Computer system of bank has credit process and debit process

```
Credit process               Debit process
shared int balance;          shared int balance;
private int amount;          private int amount;

balance += amount;           balance -= amount;


lw  $t0,balance
                             lw  $t2,balance
                             lw  $t3,amount
                             sub $t2,$t2,$t3
                             sw  $t2,balance
lw  $t1,amount
add $t0,$t0,$t1
sw  $t0,balance
```

computer architecture and organization 38

The Critical Section Problem

- n processes all competing to use some shared data
- Each process has code segment, called *critical section*, in which shared data is accessed.
- **Problem** – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Structure of process

```
while (TRUE){
  entry_section();
  critical_section();
  exit_section();
  remainder_section();
}
```

computer architecture and organization 39

Solution to Critical Section Problem

- Correct solution must satisfy
  - **Mutual Exclusion** - If process *Pi* is executing in its critical section, no other process can be executing in its critical section
  - **Progress** - Processes not in their critical section may not prevent other processes from entering their critical section
  - **Deadlock freedom** - If there are one or more processes that want to enter their critical sections, the decision of which process is allowed to enter may not be postponed indefinitely
  - **Starvation freedom** - There must be a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request
- Initial attempts
  - Only 2 processes, P_0 and P_1
  - Processes share some variables to synchronize their actions

computer architecture and organization 40

Attempt 1 - Strict Alternation

Process P_0

```
shared int turn;

while (TRUE){
  while (turn!=0);
  critical_section();
  turn = 1;
  remainder_section();
}
```

Process P_1

```
shared int turn;

while (TRUE){
  while (turn!=1);
  critical_section();
  turn = 0;
  remainder_section();
}
```

- Satisfies mutual exclusion, but not progress

computer architecture and organization 41

Attempt 2 - Warning Flags

Process P_0

```
shared int flag[2];

while (TRUE){
  flag[0] = TRUE;
  while (flag[1]);
  critical_section();
  flag[0] = FALSE;
  remainder_section();
}
```

Process P_1

```
shared int flag[2];

while (TRUE){
  flag[1] = TRUE;
  while (flag[0]);
  critical_section();
  flag[1] = FALSE;
  remainder_section();
}
```

- Satisfies mutual exclusion, but contains a deadlock

computer architecture and organization 42

Attempt 3 - Peterson's Algorithm

Process P_0

```
shared int turn;
shared int flag[2];

while (TRUE){
  flag[0] = TRUE;
  turn = 0;
  while (turn==0&&flag[1]);
  critical_section();
  flag[0] = FALSE;
  remainder_section();
}
```

Process P_1

```
shared int turn;
shared int flag[2];

while (TRUE){
  flag[0] = TRUE;
  turn = 0;
  while (turn==0&&flag[1]);
  critical_section();
  flag[0] = FALSE;
  remainder_section();
}
```

- Correct solution

computer architecture and organization 43

Synchronization Hardware

- Why not disable interrupts?

```
while (TRUE){
  disableInterrupts();
  critical_section();
  enableInterrupts();
  remainder_section();
}
```

- Unwise to give user the power to disable interrupts
- Does not work on multiprocessor systems

computer architecture and organization 44

Synchronization Hardware (cont.)

- Test-And-Set-Lock (`tsl`) instruction
  - loads contents of memory cell in register **and**
  - writes 1 into memory cell
- Executed **atomically**. If 2 processors execute `tsl` simultaneously, they will be executed sequentially in arbitrary order
  - implemented by *locking* memory bus

```
L: tsl  $t0,lock     # $t0=lock; lock=1
   bne  $t0,$zero,L  # if ($t0!=0) goto L
   ...               # critical section
   sw   $zero,lock   # lock=0
```

computer architecture and organization 45

Semaphores

- Discussed synchronization mechanisms are too low level
- **Semaphore -** integer variable which can only be accessed via two atomic operations

wait(S): if (S==0) "put current process to sleep";
S = S-1;

signal(S): S = S+1;
if ("processes are sleeping on S")
"wake one up";

computer architecture and organization 46

Example: Critical Section With *n* Processes

```
shared semaphore mutex = 1;

while (TRUE){
  wait(&mutex);
  critical_section();
  signal(&mutex);
  remainder_section();
}
```

computer architecture and organization 47

Example: Enforce Certain Order

- Execute f1() in P_1 only after executing f0() in P_0

Process P_0

```
shared semaphore sync=0;

f0();
signal(&sync);
```

Process P_1

```
shared semaphore sync=0;

wait(&sync);
f1();
```

- Question: Three processes P_1, P_2, P_3 print the string abcabcabca... P_1 continuously prints a, P_2 prints b, P_3 prints c. Give code fragments.

computer architecture and organization 48

Semaphore Implementation

- Wait and signal must be atomic (do you see why?)
- Suppose process is represented by structure

```
struct process{
  int state;    /* ready, waiting or running    */
  unsigned pc;  /* program counter              */
  struct process *next, *prev; /* list of proc. */
}
```

- Define semaphore as

```
struct semaphore{
  int value;
  struct process *list;
}
```

computer architecture and organization 49

Semaphore Implementation (cont.)

```
void wait(struct semaphore *s)
{
  disableInterrupts();
  s->value--;
  if (s->value < 0){
    /* remove calling process from ready queue */
    /* insert into s->list                     */
  }
  enableInterrupts();
}
```

```
void signal(struct semaphore *s)
{
  disableInterrupts();
  s->value++;
  if (s->value <= 0){
    /* remove a process from s->list */
    /* insert into ready queue       */
  }
  enableInterrupts();
}
```

computer architecture and organization 50

Deadlock and Starvation

- Deadlock - two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes.
- Let *S* and *Q* be two semaphores initialized to 1

P_0	P_1
wait(*S*);	*wait*(*Q*);
wait(*Q*);	*wait*(*S*);
⋮	⋮
signal(*S*);	*signal*(*Q*);
signal(*Q*)	*signal*(*S*);

- Starvation – indefinite blocking. A process may never be removed from semaphore queue. Use FCFS.

computer architecture and organization 51

Classical Synchronization Problems

- Producer-Consumer Problem (cf. Linux pipe mechanism)
  - producer and consumer share fixed-size buffer
  - cannot consume from an empty buffer
  - cannot produce in a full buffer
  - producer / consumer cannot access buffer data structure simultaneously
- Solution: use three semaphores
  - `full`: counts number of full buffer slots (initial value?)
  - `empty`: counts number of empty slots (initial value?)
  - `mutex`: controls access to buffer

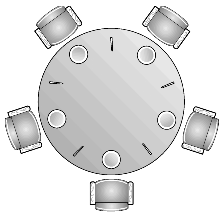
computer architecture and organization 52

Producer-Consumer Problem (cont.)

```
#define N ... /* buffer size */

semaphore mutex=1, full=0, empty=N;

void producer(void)        void consumer(void)
{                          {
  item i;                    item i;

  while (TRUE){              while(TRUE){
    produce_item(&i);
    wait(&empty);              wait(&full);
    wait(&mutex);              wait(&mutex);
    add_item(&i);              remove_item(&i);
    signal(&mutex);            signal(&mutex);
    signal(&full);             signal(&empty);
                               consume_item(&i);
  }                          }
}                          }
```

computer architecture and organization 53

Dining Philosophers Problem



- Shared data

```
semaphore fork[5]; /* Initially 1, meaning */
                   /* “on the table”       */
```

computer architecture and organization 54

Dining Philosophers Problem (Cont.)

```
semaphore fork[5];
fork[0] = fork[1] = fork[2] = fork[3] = fork[4] = 1;

void philosopher(int i)
{
  while (TRUE){
    think();
    wait(&fork[i]);
    wait(&fork[(i+1)%5];
    eat();
    signal(&fork[i]);
    signal(fork[(i+1)%5];
  }
}
```

- Contains possible deadlock (can you see it?)

Dining Philosophers Problem (Cont.)

- To avoid deadlock, pick up forks in increasing order

```
void philosopher(int i)
{
  while (TRUE){
    think();
    wait(&fork[MIN(i,(i+1)%5)]);
    wait(&fork[MAX(i,(i+1)%5)];
    eat();
    signal(&fork[MIN(i,(i+1)%5)]);
    signal(fork[MAX(i,(i+1)%5)];
  }
}
```


Threads

- Concurrently executing processes sometimes form a convenient programming model
- However, hostile processes have to be protected from each other
  - IPC is difficult (in Linux, processes cannot directly communicate via shared variables)
  - time for a context switch can be large
- **Threads**
  - processes running in the same address space
  - sometimes called *lightweight processes*

pthreads

- `pthreads` functions
  - `pthread_create`: creates a new thread
  - `pthread_exit`: exits a thread
  - `pthread_join`: wait for another thread to terminate
  - `pthread_mutex_init`: initialize a new mutex
  - `pthread_mutex_lock`: lock a mutex
  - `pthread_mutex_unlock`: unlock a mutex
- Java also supports threads
  - Synchronization mechanism: **monitors**

Tiny Threads Library

- Process creation and passing control
```
int new_thread(int (*start_add)(void), int stack_size);
void release(void);
```
- Communication
```
int get_channel(int number);
int send(int cd, char *buf, int size);
int receive(int cd, char *buf, int size);
```
- Tiny Threads Library is **non-preemptive**

Function Calls on the 80x86

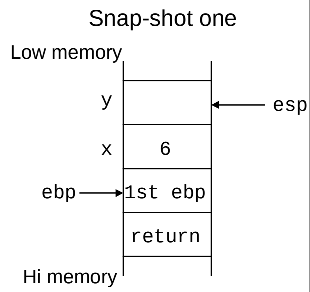
- `call` pushes return address on stack and jumps it
- `ret` pops return address from stack and jumps to it
- ebp register is used to find return address

Function Calls (cont.)

- Steps when C-function is called
  - caller evaluates argument expressions and pushes on stack
  - call function
  - push ebp on stack and copy esp to ebp
  - decrement esp to make room for local variables (like in MIPS, stack grows from low to high addresses)
- Steps when function terminates
  - copy ebp to esp
  - popd top of stack (old ebp) into ebp register
  - ret from function
  - caller increments esp to discard arguments

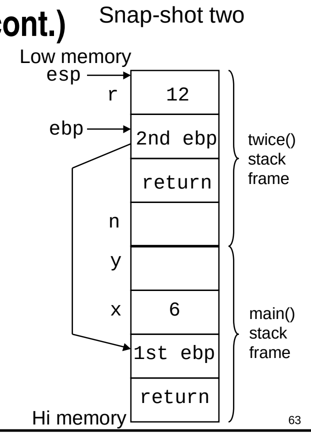
Function Calls (cont.)

```
main()
{
  int x, y;
  x = 6;
  /* snap-shot one */
  y = twice(x);
}

twice(int n)
{
  int r;
  r = 2*n;
  /* snap-shot two */
  return r;
}
```



Function Calls (cont.)

```
main()
{
  int x, y;
  x = 6;
  /* snap-shot one */
  y = twice(x);
}

twice(int n)
{
  int r;
  r = 2*n;
  /* snap-shot two */
  return r;
}
```



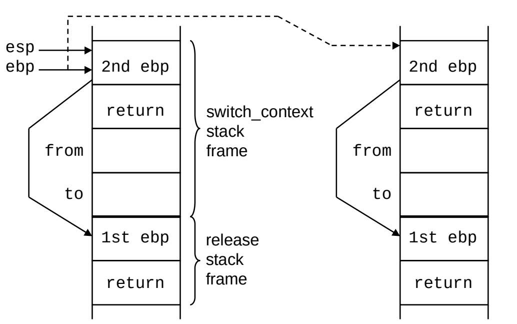
Context Switching

```
struct context{
  int ebp;
  char *stack;
  struct context *next;
  struct context *prev;
};
```

```
int release(void)
{
  if (thread_count<=1)
    return 0;
  current = current->next;
  switch_context(current->prev,
                 current);
  return 1;
}
```

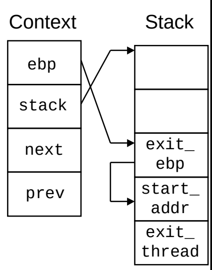
```
static switch_context(struct context *from, struct context *to)
{
  __asm__
  (
    “movl 8(%epb),%eax\n\t”
    “movl %ebp,(%eax)\n\t”
    “movl 12(%ebp),%eax\n\t”
    “movl (%eax),%ebp\n\t”
  )
}
```


Context Switching (cont.)



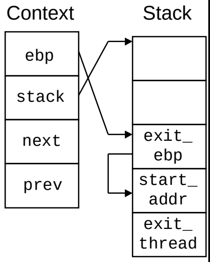
Creating and Deleting Threads

```
int new_thread(int (start_addr)(void),
               int stack_size)
{
  struct context *ptr;
  int esp;

  ... Check yourself ...

  esp =
   (int)(ptr->stack+stack_size-4);
  *(int *)(esp = (int)exit_thread;
  *(int *)(esp-4) = (int)start_addr;
  *(int *)(esp-8) = esp-4;
  ptr->ebp = esp-8;

  if (thread_count++)
    ... Check yourself ...
  else {
    ptr->next=ptr->prev=current=ptr;
    switch_context(&main_thread,
                   current);
  }
}
```



Creating and Deleting Threads (cont.)

```
static void
exit_thread(void)
{
  struct context dump, *ptr;

  if (--thread_count){
    ... Check yourself ...
    switch_context(&dump,current);
  }
  else {
    free(current->stack);
    free(current);
    switch_context(&dump,
                   &main_thread);
  }
}
```

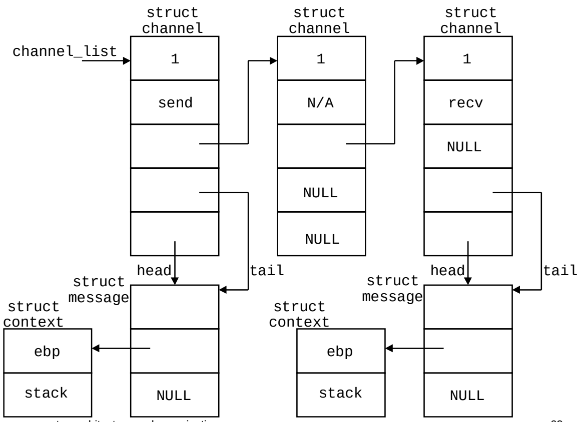


Communication

- Communication is **rendez-vous**
  - If send occurs before receive, sender is blocked until receiver arrives at same point (and vice versa)
  - Implemented by removing thread context from the ready queue
- New data structures:

```
struct channel {
  int number;
  int sr_flag;
  struct channel *link;
  struct message *message_list;
  struct message *message_tail;
};
```

```
struct message {
  int size;
  char *addr;
  struct message *link;
  struct context *thread;
};
```

Communication (cont.)

- `get_channel` function
  - first call: new channel created
  - subsequent calls: returns channel descriptors

```
int get_channel(int number)
{
  struct channel *ptr;

  for (ptr=channel_list; ptr; ptr=ptr->link)
    if (ptr->number==number)
      return((int)ptr);

  ... Allocate and initialize new channel struct ...
  ... Return address of new struct as channel descriptor ...
}
```


Communication (cont.)

- `send` and `receive` are fully symmetrical
  - can be implemented using auxiliary function `rendezvous` which has one extra parameter (direction of data transfer)

```
int send(int cd, char *addr, int size)
{
  return(rendezvous((struct channel *)cd,addr,size,1);
}
```

```
int receive(int cd, char *addr, int size)
{
  return(rendezvous((struct channel *)cd,addr,size,2);
}
```


Communication (cont.)

```
static int rendezvous(struct channel *chan, char *addr,
                      int size, int sr_flag)
{
  struct message *ptr;
  int nbytes;

  if (sr_flag == 3-chan->sr_flag){
    /* There is a communication partner of opposite type */
    ... Reinsert blocked thread into ready queue ...

    ... Calculate number of bytes to transfer and ...
    ... copy it into struct message so that communication ...
    ... partner can access it ...

    ... Copy data and reset send/recv flag if required ...
    return(nbytes);
  }
  else {
    /* There is no communication partner of opposite type */
```

See next page

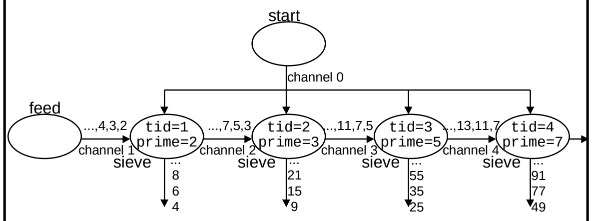
```
  }
}
```

Communication (cont.)

```
static int rendezvous(struct channel *chan, char *addr,
                      int size, int sr_flag)
{
                     See previous page
    /* There is no communication partner of opposite type */
    ... Allocate new msg struct and insert in list of msgs ...

    ... Remove current thread from ready queue and link ...
    ... under msg struct ...

    if (--thread_count){
      current = current->next;
      switch_context(ptr->thread, current);
    }
    else
      switch_context(ptr->thread, &main_thread);

    /* When blocked thread is resumed, it will return here */
    nbytes = ptr->size;
    free(ptr);
    return(nbytes);
  }
}
```


Sieve of Eratosthenes

- Idea
  - Organize threads in a pipeline
  - Thread *i* is responsible for sifting out multiples of *i*-th prime
  - Numbers that “make it through the pipeline” are prime



- Figure out the code yourself (threads are created on demand)