CS 520, *Operating Systems Concepts*

Lecture 1

Introduction and

Process Creation and Synchronization

Reading

- Required text: **ZyBooks** *Operating System Concepts*, *11th Edition*, by A. Silberschatz, P. B. Galvin, and G. Gagne
- Additional recommended reading:
  - *Modern Operating Systems* by A. Tanenbaum
  - *Operating System Design: The XINU Approach* by D. Comer

Why Should You Study Operating Systems...

...when you may never write one?

Because

- It will make you understand *fundamental* issues of computer science that are applicable to many other fields (data communications, for example, or even factory management)
- It is necessary to deal with *all* topics of computer security
- It is at the *heart* of Cloud Computing
- It is interesting
- In fact, you may be writing one (open source)!

An Operating System is...

...a set of programs that provides the user of a computer (e.g., an application programmer, an end-user using an application, or even another computer) with the interface to the computer's hardware by supporting a set of *services*.

Operating System Services

- Program execution
- Input/Output (I/O) operations
- File-system support
- Interprocess Communications
- Error detection
- Resource allocation
- Accounting
- Protection

What is “Services?”

- Services are invoked by users calling the OS library of procedures.
- The Operating System Kernel can be seen as a library of objects whose methods are called… directly by the CPU!

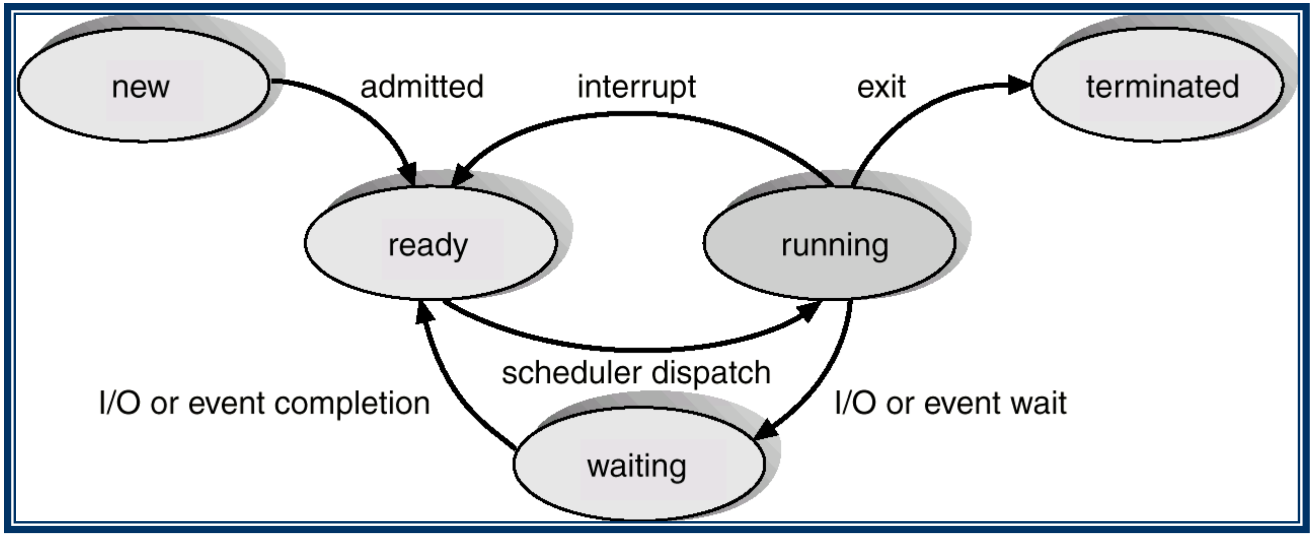
A Process

- Modern operating systems (we will discuss the history and evolution of computing later) support *multiprogramming*—that is an ability to execute several programs concurrently on one CPU or simultaneously on several CPUs
- A *process* is a program in execution (a useful metaphore):
  - A program is a cookbook
  - A CPU is a cook
  - I/O devices are cooking utensils
  - A process is making a dish described in the cookbok

Execution of Processes

- A process is like a puppet—a CPU puts it on its hand and makes it alive; then puts it away and picks up another process. (This is called *context switching*.) With that it remembers the state of each process, so when the process resumes it is *unaware* of its having been put away!
- Over its life, a process goes through a basic set of states

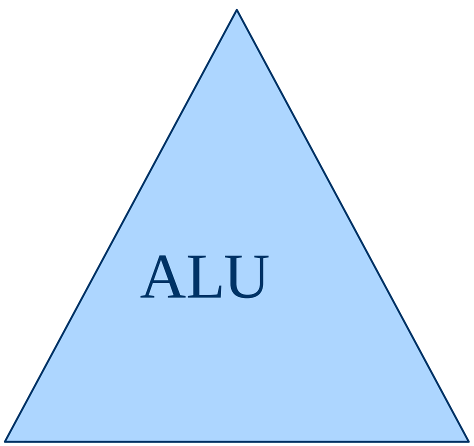
Process States



What we are going to do in the rest of the semester is *divide and conquer*

- We will deal with process synchronization, resource management, I/O devices, memory management, file systems, etc., but *one problem at a time*.
- Today we will recall the basic principles of computer organization and then
- Move on to process- creation, synchronization and mutual exclusion—this is, by far the hardest problem (it will be easier to study later!), but it is the most interesting one and...
- ...it does *not* require much pre-requisite knowledge, just the understanding that a process may be interrupted any time between two instructions

CPU



Registers:

General Registers
Program Counter (PC)
User Stack Pointer (SP)
System Stack Pointer (SP)
Status Register

How CPU Works (first approximation)

A tight loop:

Move	@R1	R2
ADD	R2	R3

```
While TRUE
  {
      Fetch an instruction pointed by the PC;
      Advance the PC to the next instruction;
      Execute the instruction
      Update the Status Register;
   }
```

The process's stack and the procedure call

Main line of the process code

100000	LOAD	R1	@20002
100010	LOAD	R2	@20010
100020	STORE	R1	@SP
100030	ADD	SP	#-4
100040	STORE	R2	@SP
100050	ADD	SP	#-4
100060	ADD	SP	#-8
100070	JPR	10000000	
100080	ADD	SP	#16
100090	…	…	…

Stack Frame

Saved PC (100080)
Internal variable 2
Internal variable 1
Parameter 2
Parameter 1

The procedure code

10000000	**LOAD**	**R1**	**@(SP + #20)**
…	…	…	…
10005000	**RTP**		

CPU: First Approximation

A tight loop:

```
While TRUE
  {
    Fetch an instruction pointed to by the PC;
    Advance the PC to the next instruction;
    Execute the instruction;
    If an exception has been raised
      {
        Save the PC on the process stack (@SP)
        PC = Interrupt_Vector[exception];
      }
  }
```

Process Stack
Saved PC

Interrupt routine

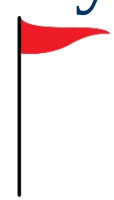
DIS
...
RTI

Privileged Instructions

- Instructions that deal with processing interrupts, changing the status register, performing memory management, and the like are mission-critical.
- Critical instructions, by a long-established convention, require CPU processing in a special—***supervisory*** or ***system***—mode
- A CPU may also have a special set of registers, reserved for the system mode. A separate, **supervisory stack pointer** points to a separate stack
- All exception processing is performed in supervisory mode (A *Unix* example) Each process therefore actually has two stacks: a ***user stack*** and a ***system stack***.

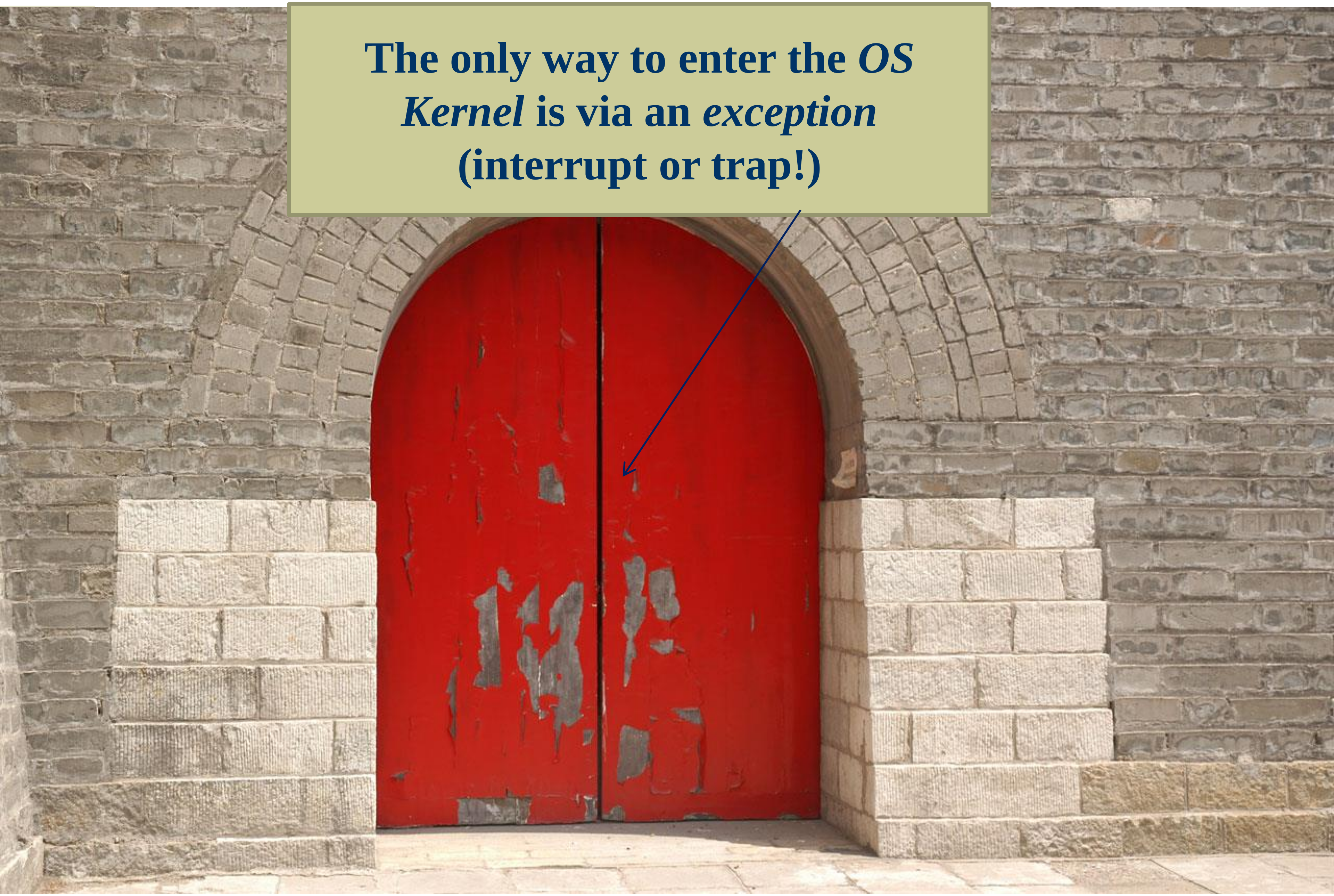
CPU Mode (an essential security feature)

- To ensure that *only* the OS can execute system code (interrupt processing, memory manipulation, etc.), modern CPU execute the system and user code in different modes
- The mode is typically indicated by a flag in the STATUS register

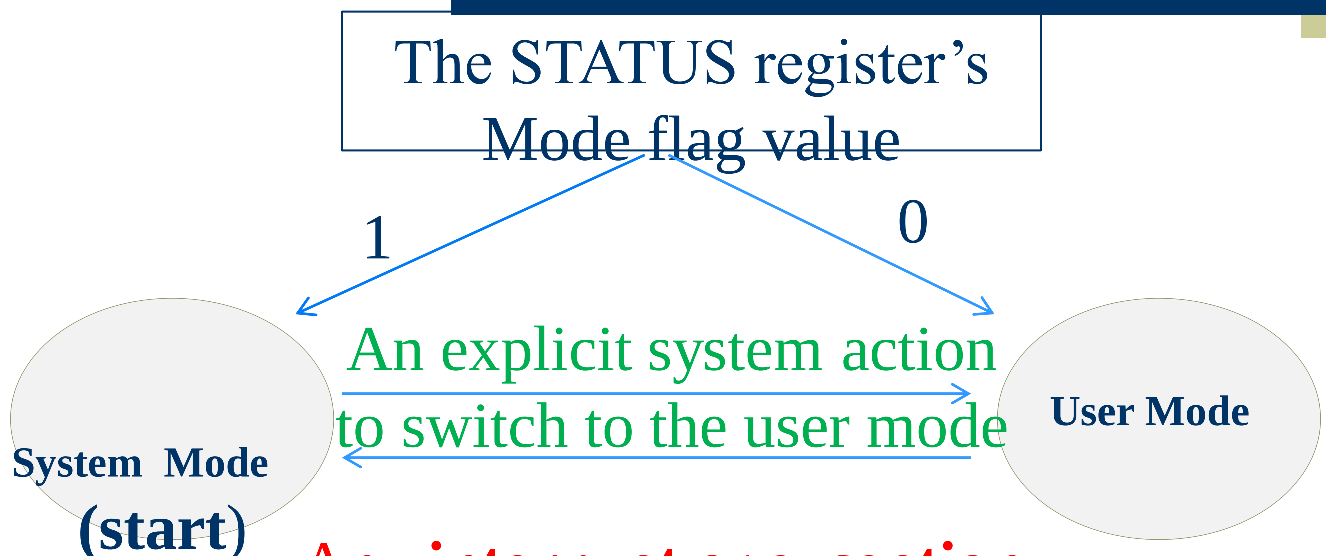


00100100`0

The only way to enter the *OS Kernel* is via an *exception* (interrupt or trap!)



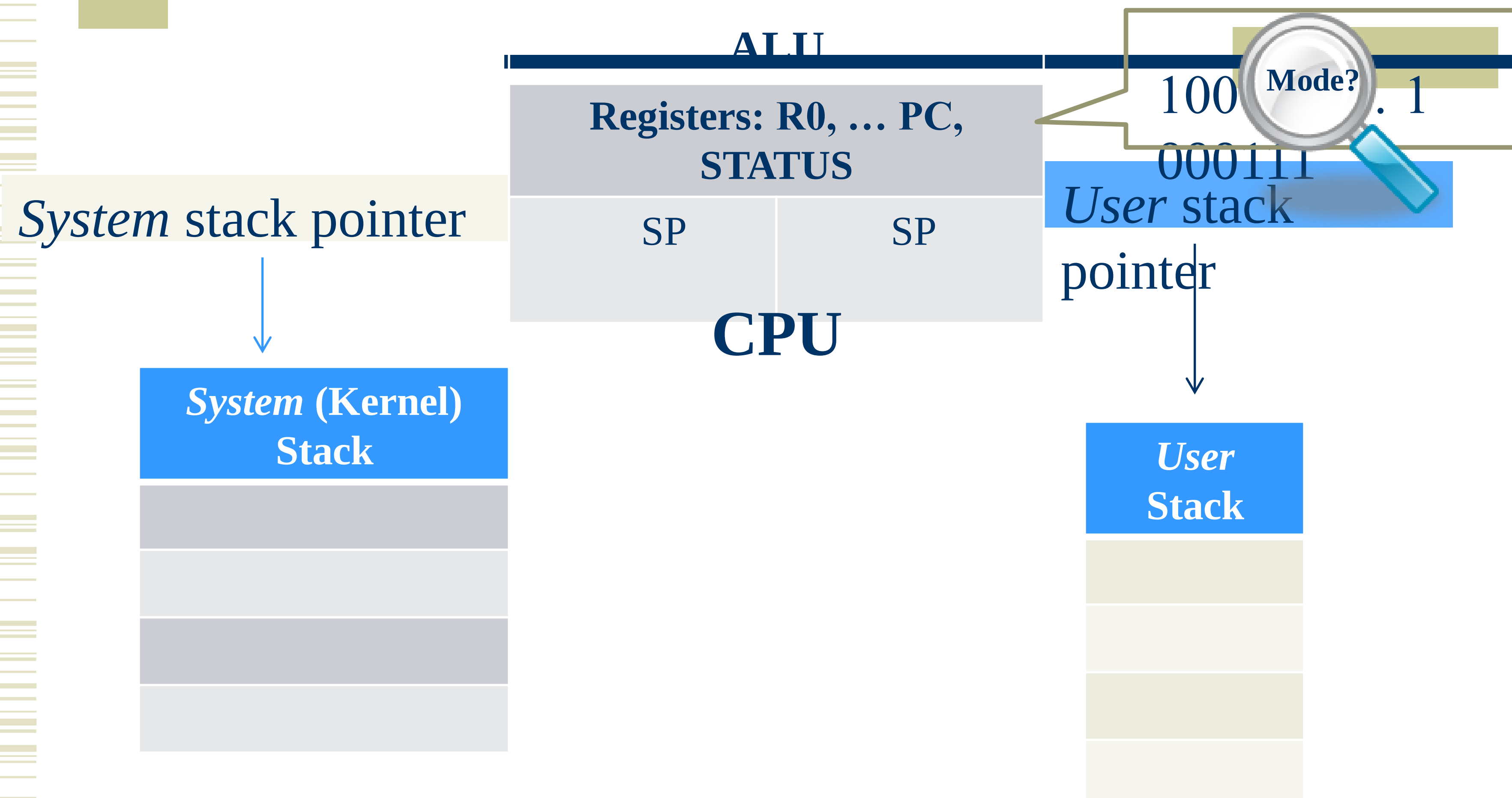
The *CPU mode* state machine



Any **interrupt** or **exception** (including that caused by an attempt to execute a **privileged instruction**)

The System Mode and the User Mode are associated with separate stacks

The modified CPU and the two process stacks



The CPU loop—the final version

```
While TRUE
  {
     Fetch an instruction pointed to by the PC;

     If the instruction is valid AND
        the instruction is appropriate for the present mode AND
        the parameters are valid for the operation
          {
               Advance the PC to the next instruction;
               Execute the instruction;
           }
      else
         raise an appropriate exception;
     If (an exception #x has been raised) OR
        (an interrupt #x has been raised) AND interrupts are enabled
       {
          Save the STATUS register and PC on the system stack (@SP);
         Switch to the system mode;
         PC = Interrupt_Vector[x];
       }
  }
```

What happens next?

- So far, it worked exactly like a procedure call (with the return address saved on stack), except that the call was initiated by CPU itself—not the process!
- The interrupt routine typically
  - Disables interrupts to perform some critical operations
  - Enables interrupts and calls other routines
  - Exits by executing the *Return from Interrupt (RTI)* instruction, which *pops* the stack and replaces the PC so as to return to the interrupted program:

    PC = Return PC

Exceptions: Interrupts vs Traps

- The *interrupts* are caused, asynchronously with the program execution, by *external* events (I/O request completion, input arrival, clock)
- Yet, the same exception process, can be triggered synchronously—by an instruction, while, for example,
  - Referring to a wrong address (bus error) as in
    MOVE R1, FFFFFFFF
  - Performing a wrong arithmetic operation (e.g., dividing by *0*)
  - Attempting to execute an undefined (or illegal) instruction
  - Executing (intentionally) a *trap* instruction

An example

- We request a "service," say by calling

```
Write(record, file)
```

- The actual code for the 'write' routine would look like that:

```
void write (byte record[], file_handle file)
    {
        #0733   /* Trap instruction */
    }
```

This Is a Major Security Feature

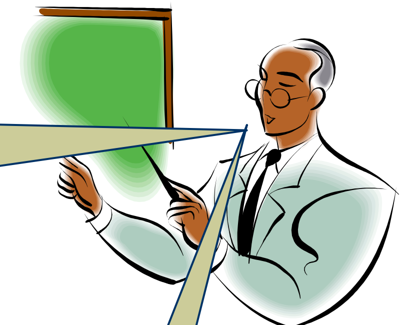
- When we get into a taxi, we tell the driver where we want to go—we don't drive it ourselves!
- When we use OS, we tell it what we want to do, but we don't get the direct access to systems resource.

Creating a child process in Unix (or Linux)

P_1

Fork?

You are the parent.
Your child is
P_2!



OS

P_2

Fork?

You are the child!

Fork()

```
main()
  {
     int fork_result;
    fork_result  = fork();
       if (fork_result >= 0)  /* the child has been created */
         if (fork_result  == 0)  /* child */
           New_life(); _
         else
          {
          … /* Continue old life; the child's PID is in fork_result. */
          }
       else
        …
  }
```

How many processes will be running?

```
main ()
  {
     fork();
     fork();
     fork():
   }
```

Another example

```
main()
 {
     int fork_result;
     int example = 18;
    fork_result  = fork();
        if (fork_result >= 0)  /* the child has been created */
          if (fork_result  == 0)  /* child */
           { New_life();
             example++;
             printf ("Child: %d.",  example);
           }
          else
           {
            printf ("Parent: %d.",  example);
           }
 }
```

Race Conditions and Synchronization

- Processes that execute concurrently (or in parallel) often need to share common storage
  - Consider the *Producer-and-Consumer)* problem where one process produces items and fills out a (bounded) array, and the other consumes these items and frees the array space:

out

Race Conditions

- Reading data concurrently is alright, but writing is problematic. Suppose $x = 2$, and two processes *P1* and *P2* execute :

```
MOVE @x R1
ADD  R1 #1
MOVE R1 @x
```

Race Conditions: Unpredictable Outcome Because of Preemption

P_0:

```
MOVE @x R1
ADD R1 #1
MOVE R1 @x
```

P_1:

```
MOVE @x R1
ADD R1 #1
MOVE R1 @x
```

x = 4

```
MOVE @x R1
ADD R1 #1
MOVE R1 @x
```

```
MOVE @x R1
ADD R1 #1
MOVE R1 @x
```

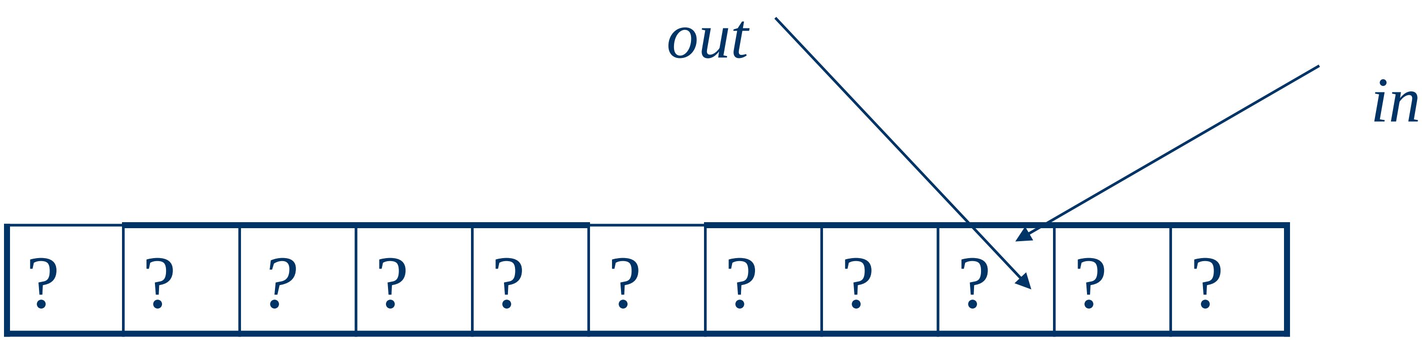
x = 3

Criteria for a Solution

1. **Only** one process may be inside the critical section
2. No process outside the critical section may block access to it
3. No process should be caused (by the synchronizing algorithm itself) to wait forever to enter the critical section (bounded waiting)

An Example: *Producer-and-Consumer* (Bounded Buffer) Problem

- accessing the *in* and *out* values can be done only in a critical section
- If *in* = *out*, then either the buffer is full or empty; unless *buffer_size – 1* entries are used, it is essential to maintain a *count* variable



Critical Section Access—Disabling Interrupts

- One way to provide *mutual exclusion* is to disable interrupts (and thus context switching), but
  - It cannot be done for long (or some interrupts will be lost)
  - User processes are not allowed to do that
  - I/O may be needed while in a critical section (and it will never be completed with interrupts disabled)
  - It does not work in a multi-processor system, anyway!

A Hardware-supported Solution: *Test and Set Lock (TSL)*

- The *TSL* instruction reads a value at a memory location into a register and *then* sets it to *1*—both operations combining into an *atomic* one:

```
Enter_Critical:
    TSL A @lock          | A =lock; lock = 1.
    CMP A #0             | Was it 0?
    JNZ Enter_Critical| Cycle until it is 0
    RTS                  | Return to caller

Exit_Critical:
    MOVE @lock #0
    RTS
```

One software “solution”: Strict Alternation

P_0:

```
while (TRUE)
   {
     <non-critical section>
     while (turn == 1);
     <critical section>
     turn = 1;
     <non-critical section>
   }
```

P_1:

```
while (TRUE)
   {
     <non-critical section>
     while (turn == 0);
     <critical section>
     turn = 0;
     <non-critical section>
   }
```

More precisely:

```
int turn = 0;                /* which process? */

void Enter_Critical (int process) /* 0 or 1 */
     int other;
 {
    other = 1 - process;

    while (turn == other);  /* wait for the other */
  }

void Exit_Critical (int process) /* 0 or 1 */
  int other;
 {
    other = 1 - process;
    turn = other;
 }
```

Are the Criteria Met?

- No more than one process may be inside the critical section. TRUE.
- No process outside the critical section may block access to it. FALSE.
- No process should wait forever to enter the critical section. TRUE as long as there are only two processes, and none is stuck.

Peterson’s Solution (1981)

```
#define N     2  /* number of processes */

int turn;             /* which process? */
int flag [N];                 /* to signal N's interest in entering;
  initially                          initially FALSE */

void Enter_Critical (int process) /* 0 or 1 */
    int other;
 {
    other = 1 - process;
    flag[process] = TRUE;    /* show my interest */
    turn = other;            /* be kind to the other process */
    while (flag [other] == TRUE && turn == other);  /* wait */
  }

void Exit_Critical (int process) /* 0 or 1 */
 {
   flag[process] = FALSE;       /* I lost my interest */
 }
```

In Other Words

P_0:

```
while (TRUE)
  {
    Enter_Critical(0);
    ... critical ...
    Exit_Critical(0);
   ... non-critical ...
  }
```

P_1:

```
while (TRUE)
  {
    Enter_Critical (1);
    ... critical...
    Exit_Critical (1);
   ... non-critical ...
  }
```

Are the Criteria Met Now?

- No more than one process may be inside the critical section. TRUE.
  - *Proof*: Suppose they are. Then *flag = {TRUE, TRUE}*. But *turn* can be either *0*, or *1*; thus only one process executing the *WHILE* loop could have passed!
- No process outside the critical section may block access to it. TRUE.
- No process should wait forever to enter the critical section. TRUE.

A Homework Assignment Creeping

```
void Enter_Critical (int process) /* 0 or 1 */
 {
     other = 1 - process;
     flag[process] = TRUE;    /* show my interest */
```

If these statements

```
    turn = other; /* be kind to the other process */
    while (flag [other] == TRUE && turn == other);
```

are replaced with the following ones,

```
    turn = process;              /* grab it! */
    while (flag [other] == TRUE && turn == process);
```

will the algorithm work? Why (or why not)?

A Generalized Solution—the Bakery Algorithm

```
#define N     ...   /* number of processes in the system*/
int number [N];           /* dynamically assigned, initally 0 */
int choosing [N]; /* N-th process has chosen its number */

void System_Init() /* to be executed once at start */
 int i;
 {
   for (i=0; i<N; i++)
      {
        choosing[i] = FALSE;
        number[i] = 0;
      }
  }
int less (int a, b, c, d); /* defines [a,b] < [c, d] * /
 {
   if (a == c)less = (b < d)
   else less = (a < c);
   return less;
 }
```

The Bakery Algorithm (*cont.*)

```
void Enter_Critical (int process) /* 0 to N-1 */
 int t;
 {
    choosing[process] = TRUE;
    number [process] = max (number[0],...,number[N-1]) + 1;
    choosing [process] = FALSE;
    for (t=0; t<N; t++)
      {
        while (choosing[t]);/* wait for those who are choosing */
        while ( ( (number[t] != 0) &&
                less (number[t], t, number[process], process)
              ); /* wait until the lowest-numbered process
                    executes */
      }

 )
void Exit_Critical (int process)
 {
   number[process] = 0;
 }
```

Some Observations

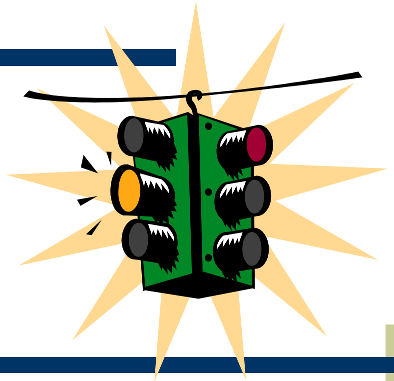
- Peterson's is an ingenious solution. Compared with the Dekker's algorithm, the only one known for 17 years before, it is much easier to understand
- Peterson's solution can be extended to any number of processes, and it works perfectly well in a distributed system, but it has the same problem as TSL does: *Busy Waiting* (also called *spinlock*)
- Busy waiting is generally unacceptable, especially on a uniprocessor—it is wasting CPU time. It is much better to cause a process to be suspended until the condition it is waiting for holds. But a limited *system-controlled* spinlock of TSL is a much better solution than *user-coded* busy waiting

Semaphores

A Semaphore S can only be accessed via one of the two atomic operations *wait(S) and signal (S).*

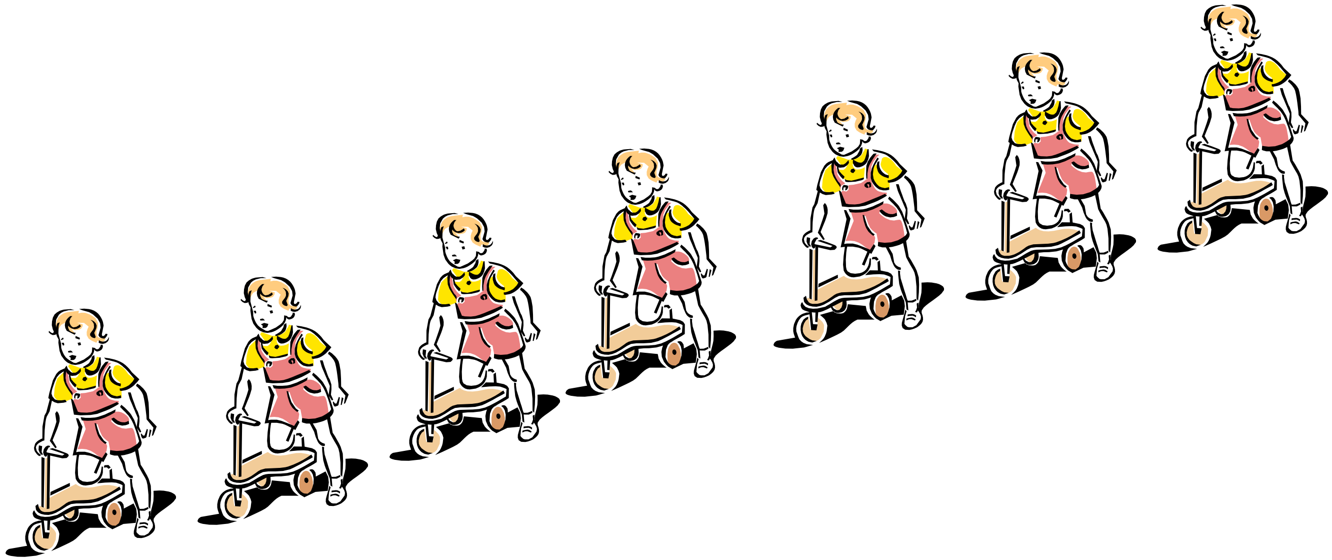
```
typedef struct
     {
       int value;
       struct process_queue_type queue;
     } semaphore;

Constructor (initialization)

  void wait (semaphore S)
     {
       S.value--;
       if (S.value < 0)
         queue_and_block(S.queue);
     }

void signal (semaphore S)
     {
       S.value++;
       if (S.value <= 0)
         advance_queue(S.queue);

     }
```

A Semaphore



Value: -6

Queue:

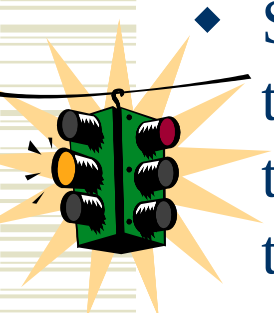


Mutual Exclusion with Semaphores: Example

Several processes share a semaphore *mutex*, where *mutex.value* is initialized to 1. A process' code would look like.

```
... non-critical section ...
wait (mutex);
... critical section ...
signal (mutex);
... non-critical section ...
```

The Two Purposes of Semaphores



- Semaphores can be used for mutual exclusion, as shown in the previous example. This is equivalent to waiting for a traffic light so as to avoid a collision. Semaphores used for that are typically initialized to 1. (They are called *binary sempahores*.)

- Semaphores can also be used for *synchronization*. This is equivalent to waiting for a date (rather than avoiding something). The initial values of such semaphores can be anything. See excercise #8 in the homework.

Semaphores: Dangers

- It is fairly easy to make a mistake when programming with semaphores. They are *not* trivial to use. (We will learn about a few typical problems—*deadlock* and *starvation* while doing the homework.)
- The types of mistakes involving semaphores are particularly dangerous—they can wreck the whole system; often, they are very hard to detect, too.

The Producer/Consumer (Bounded Buffer) Problem
Solution with Semaphores

```
#define N = ...;
semaphore mutex=1, empty = N; full = 0;
int in=0, out=0;
```

```
while (TRUE)
  {/* Producer */
    produce_item(&item);
    wait(empty);
    wait(mutex);
      buffer[in] = item;
      in = (in+1) % N;
    signal(mutex);
    signal(full); /* NB! */
  }
```

```
while (TRUE)
  {/* Consumer */
    wait(full);
    wait(mutex);
      item = buffer[out];
      out = (out+1) % N;
    signal(mutex);
    signal(empty);
    consume_item(&item);
  }
```

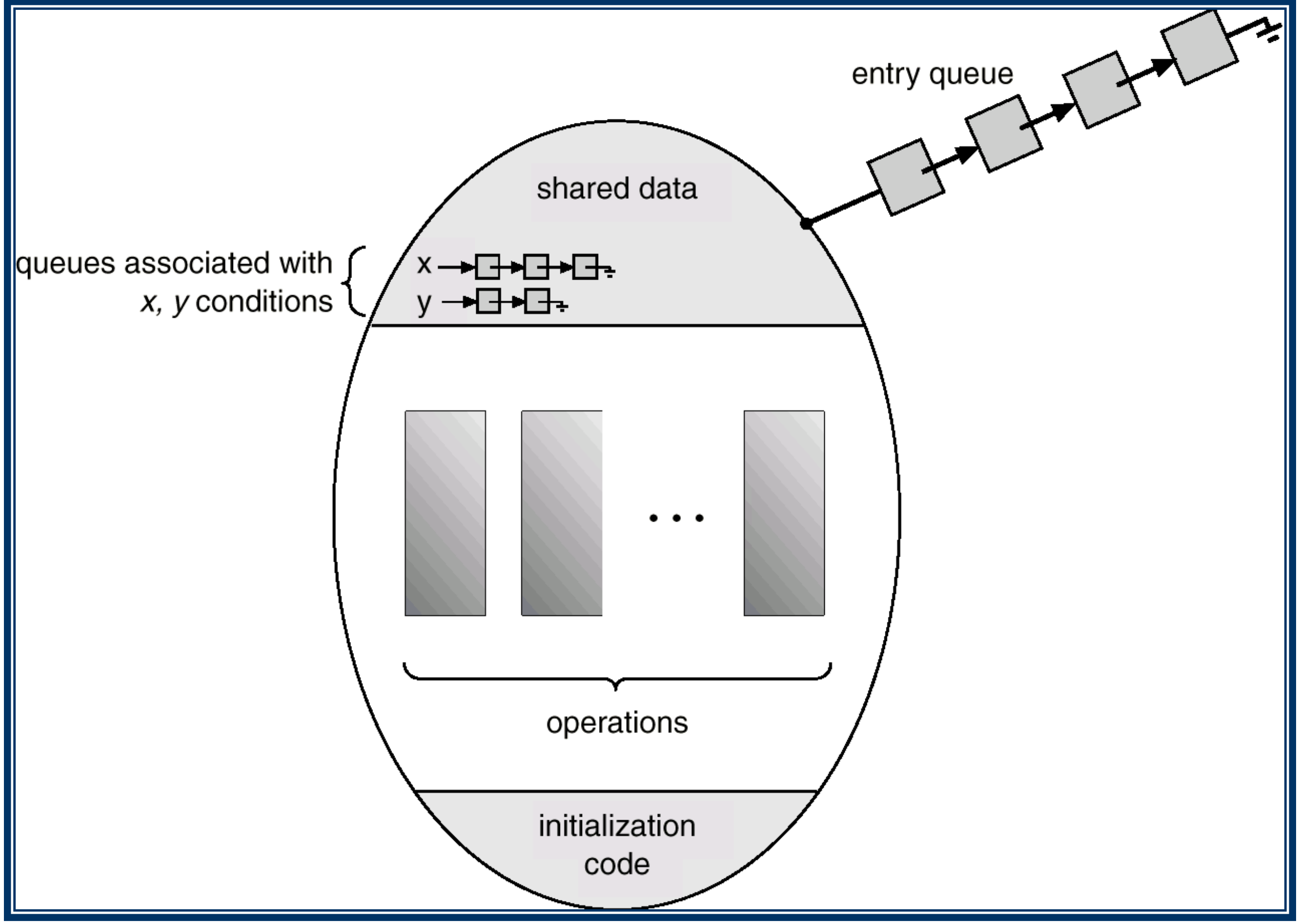
So, what *can* be done?

- A programmer's job has been traditionally made easier by compilers
- Semaphores are an operating system construct. What is needed to hide them is a new language construct
- Such a construct, called *monitor*, was introduced (almost simultaneously) by Brinch Hansen and Hoare

Monitors

- A monitor is, first of all, a *class* (that is a data structure that defines operations [*methods*] on its data). Classes are *instantiated* as *objects*.
- One aspect of the semantics of a monitor is *assumption* of concurrent execution and, consequently, specific support of mutual exclusion: only one process at a time may invoke a monitor’s method
- Another aspect is the introduction of *condition* variables, which have generic *wait* and *signal* operations

Monitors


entry queue
shared data
queues associated with
x, y conditions
x
y
operations
initialization
code

Conclusions and Problems

- Semaphores and monitors are constructs for providing 1) mutual exclusion and 2) synchronization to concurrent processes
- Both semaphores and monitors are equal in terms of the problems they can solve
- Monitors provide help to programmers in that the compiler protects them from making certain types of errors (such as permuting semaphore operations)
- Both constructs were developed with a uniprocessor (or, at most, multiprocessors with shared memory) in mind
- To deal with distributed computing, a different mechanism (message passing) is needed—we will discuss it later.