Previous class...

• Process state transition
  – Ready, blocked, running
• Call stack
  – Calling convention
• Context switch
  – Process switch
• Interrupt vs. exception vs. signal
• Mode switch
  – Interrupt/exception handling
• fork() and Shell
The three basic process states and the four transitions between them

Three states: Running, Blocked, Ready
(1) Running -> Blocked: I/O; wait for a lock
(2) Blocked -> Ready: I/O is ready; the lock is acquired
(3) Ready -> Running: a ready process is picked to run
(4) Running -> Ready: time slice used up
What is a Call Stack?

A stack data structure that stores the information for the active function calls; it consists of stack frames, each of which corresponds to an active function call.
What is Execution Context? Give two examples that are part of the context

The contents of the CPU registers at any point of time.

**Program counter**: a register that points to the instruction to be executed

**Call stack pointer**: a register that points to the top of the call stack, e.g., esp in x86 CPU
What is Context Switch? When does it occur?

Also called process switch. Suspending one process and resuming another.

The current process has used up its time slice, blocks (due to I/O or requesting lock), or exits.
What is Mode Switch? When does it occur?

Program execution switches between different CPU modes
- **User** -> **Kernel**: system calls, interrupts, exceptions
- **Kernel** -> **User**: finish handling of system calls / interrupts / exceptions
Previous class...

Difference between Interrupts, Exceptions, and Signals. Give examples

**Interrupts:** due to hardware; e.g., timer, I/O devices

**Exceptions:** due to software; e.g., breakpoints inserted by debugger, divide-by-zero, null pointer exceptions

**Signals:** due to exception handling or signaling API calls; e.g., SIGSEGV (due to illegal memory access)
How does a computer respond to a keystroke?

**Hardware part:** an interrupt is generated by the keyboard, and CPU automatically sets the program counter to the corresponding interrupt handler (recall IDT[n])

**Software (Kernel):** the interrupt handler executes the device driver for the keyboard, which reads the character from the keyboard buffer to the kernel-space buffer

**Software (User and Kernel):** the process that waits for the input is waken up (blocked -> ready), and the kernel copies the character from the kernel-space buffer to the user-space buffer
How does the system respond to divide-by zero (e.g., a/0) in user code?

1. When a divide-by-zero operation is performed, an exception (also called s/w interrupt or trap) is triggered.
2. To handle the exception, execution switches to the kernel mode.
3. The kernel handles the exception by generating a SIGFPE signal for the faulty process.
4. After exception handling, execution switches back to the user mode.
5. The signal handler is invoked (the default handler is implemented and registered by libc; it terminates the process).
Implement a simplest Shell using fork(), exec(), wait()

fork(): create a child process.
exec(): replace the execution of the current program with the designated one
wait(): suspend the calling process until any of its child process exits
What is a Zombie Process? Why is it harmful? Why is it necessary?

When a process exits, it becomes a zombie unless it is waited for by its parent or its parent has explicitly expressed no interest in the exit state.

A zombie process occupies precious kernel resources, such as PCB.

The PCB stores the exit value of the zombie process.
Outline

• IPC for passing data
  – Pipe, FIFO, Message Queue, Shared Memory

• Concepts
  – Race condition, critical section, and mutual exclusion

• Sync. with busy waiting
  – Software-based solutions
  – Hardware-assisted solutions

• Sync. without busy waiting
IPC - Pipe

- **Pipe** is an IPC technique for passing information from one process to another
- Byte stream (like water); no notion of “message boundaries”
- Example:
  - Search lines that contain some word: `cat * | grep “os”`
  - # of files in a directory: `ls | wc -l`
pipe() - details

- int pipe(int \( fd[2] \))
  - creates a new pipe and returns two file descriptors
    - \( fd[0] \): refers to the read end
    - \( fd[1] \): refers to the write end
pipe(), fork(), and close() are good friends

- pipe(): create a pipe and return fd[0] and fd[1]
- fork(): child inherits fd[0] and fd[1]
- close(): close unused file descriptors
How do pipes work under the hood?

- There is an in-memory file system, pipefs, in the kernel memory address space
- Pipefs is mounted at system initialization
- A pipe corresponds to a file created in pipefs
- Pipe() creates a pipe and returns two file descriptors
  - One for the read end, opening using O_RDONLY, and
  - One for the write end, opened using O_WRONLY
More details about pipe()

- If read() from a write-closed pipe: end-of-file
- If write() to a read-closed pipe: SIGPIPE -> write failure
- By default, blocking I/O
  - If a process attempts to read from an empty pipe, then read() will block until data is available
  - If a process attempts to write buf to a full pipe, then write() blocks until min(buf, PIPE_BUF = 4KB) is available
    - fcntl() can change it to non-blocking behavior
- If write() too large a message: the message will be split into pieces with each = PIPE_BUF (4KB in Linux) by the system
Example: passing an integer between two processes

Child:
int n = something();
write(fd[1], &n, sizeof(n));

Parent:
int n;
read(fd[0], &n, sizeof(n));

Q1: How does a writer indicate the end of data sending?
A: Either send the size to the receiver in the beginning, so the receiver know how many to expect; or send an indicator char (e.g., EOF=-1) in the end

Q2: Can a writer close the pipe immediately after sending data?
A: No. read() will return 0 immediately without receiving the data in the buffer

Q3: Then how can the writer know when to close the pipe?
A: Semaphore (to be covered soon)
Explain what happens under the hood when you run the command “ls –ls | sort”

1. Process creation: fork and exec
2. Pipe creation
3. Redirection: set ls’s STDOUT to pipe’s write end, and set sort’s STDIN to pipe’s read end
4. Scheduling:
   • ls: write->block->write->…;
   • sort: read->block->read->….
IPC – FIFO (also called Named Pipe)

- Unlike a pipe, a FIFO is not limited to be used between the parent and child processes.
Example of FIFO

```c
#define FIFO_FILE "MYFIFO"

FILE *fp;
char readbuf[80];

/* Create the FIFO if it does not exist */
mkfifo(myfifo, 0666);

fp = fopen(FIFO_FILE, "r");
fgets(readbuf, 80, fp);
printf("Received string: %s\n", readbuf);
fclose(fp); // fclose() will not delete fifo

fp = fopen(FIFO_FILE, "w")
fputs("Hello!", fp);
fclose(fp);
return(0);
```
Comparison

• Unnamed pipes
  – Purely in-memory
  – The pipe is deleted after all file descriptors that refer to it are closed
  – Have to be used between parent and child
  – Typically, 1 writer and 1 reader

• FIFOs
  – A file is created in your hard disk, but the I/O is through the kernel memory
  – Not deleted even you close the file descriptor (you need to explicitly remove it like removing a file)
  – Not limited to parent-child processes
  – There can be multiple writers and one reader
IPC – Message queues

Conceptually, a queue in kernel

<table>
<thead>
<tr>
<th>POSIX APIs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qid = mq_open(QUEUE_NAME, O_CREAT</td>
<td>O_RDONLY, 0644, &amp;attr);</td>
</tr>
<tr>
<td>qid = mq_open(QUEUE_NAME, O_WRONLY);</td>
<td>Open a queue</td>
</tr>
<tr>
<td>mq_send(mq, buffer, MAX_SIZE, 0)</td>
<td>Send a message; can specify the priority of the message</td>
</tr>
<tr>
<td>mq_receive(mq, buffer, MAX_SIZE, NULL)</td>
<td>Receive a message; can specify the priority also</td>
</tr>
</tbody>
</table>
IPC – Shared Memory

- A region of physical memory that is mapped to two (or more) processes, such that when a process updates the shared memory the other process can see the modification immediately, and vice versa
Shared Memory

Process P1

shared memory (mapped)

shared memory

Process P2

shared memory (mapped)

POSIX Shared Memory
Shared Memory

• The previous three mechanisms (Pipes, FIFOs, and Message Queues) all require mode switch when sending/receiving information (why? recall system calls), while shared memory does NOT
How to create a Shared Memory?

- **shmget**: System V API
- **shm_open**: POSIX API
- **mmap + fork** -> the easiest way to create a shared memory between processes
  - `mmap(MAP_ANONYMOUS | MAP_SHARED)`
## Compare different IPCs

<table>
<thead>
<tr>
<th>IPC method</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipes</td>
<td>Can only be used among parent and child</td>
</tr>
<tr>
<td>FIFO (named pipes)</td>
<td>Pipe is named using a string, so doesn’t have the limitation above</td>
</tr>
<tr>
<td>Message Queues</td>
<td>Supports message boundary and message types / priorities</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>Data passing doesn’t go through kernel, so it is usually the most efficient one</td>
</tr>
</tbody>
</table>
Outline

• IPC for passing data
  – Pipe, FIFO, Message Queue, Shared Memory

• Concepts
  – Race condition, critical section, and mutual exclusion

• Mutual exclusion based on busy waiting
  – Software-based solutions
  – Hardware-assisted solutions

• Synchronization without busy waiting
Race condition bug

• A race condition exists if the final program result depends on the execution sequence
• Let’s consider a counter, \( c \), that is shared by two processes

\[
++c;
\]

(1) movl c, %eax // copy to eax
(2) addl $1, %eax // increment
(3) movl %eax, c // copy back to c

• Assume \( c = 0 \) initially, and the two processes both execute “++c”
• Consider the execution sequence: after process 0 executes (1) (2), it is scheduled out; and then process 1 executes (1) – (3)?
• You get \( c = 1 \) here, while in other execution sequences you may get 2
• A concurrent program as simple as this has a race condition bug
Intuitive attempts to fix the bug

• Use atomic_add instructions
  – atomic_add supports you to do atomic increment
  – It works in that extremely simple example
  – But it is not a general solution; consider another example

```c
// critical section: withdraw $100 from account
(1) if(account >= 100)
(2)   account -= 100;
// Is it possible to withdraw $200, given account = 100?
```

• Disabling interrupts to prevent the process from being schedule out
  – Does not work well on multi-core machines
Intuitive attempts don’t work; we need a formal treatment by introducing new ideas and concepts
Critical section and mutual exclusion

• A **Critical section** is a program region that has to access shared data in a synchronized way (say, a mutual exclusion way); otherwise, race condition may occur

• **Mutual exclusion** is to make sure no two processes are simultaneously inside critical sections that access the same shared data
  – Each synchronization primitive that enforce mutex provides the two APIs: `enter_cs()` / `leave_cs()`
  – A critical section is surrounded by the two API calls
Why is Mutual Exclusion important?

// Why is it impossible to withdraw 200 given account = 100
enter_cs();
// critical section
if(account >= 100)
    account -= 100;
exit_cs();
Mutual Exclusion (Mutex)

• A good solution to mutual exclusion should satisfy
  – Mutual exclusion
  – No assumptions made about processor speed or number
  – No outside blocker
  – No infinite wait
A failed attempt: strict alternation

// process 0
while(true) {
    while (turn != 0) ; //enter_cs()
    critical_section();
    turn = 1; // leave_cs();
}

// process 1
while(true) {
    while (turn != 1) ; // enter_cs()
    critical_section();
    turn = 0; // leave_cs();
}

• What if process 0 quickly finishes one iteration and wants to execute another, while process 1 has not entered the critical section?
• It violates Condition 3 “no outside blocker”
Outline

- IPC for passing data
  - Pipe, FIFO, Message Queue, Shared Memory

- Concepts
  - Race condition, critical section, and mutual exclusion

- Mutual exclusion based on busy waiting
  - Software-based solutions
  - Hardware-assisted solutions

- Synchronization without busy waiting
Dekker’s algorithm – First solution

The two flags indicate the intention of the two processes to enter critical sections, respectively. If both intend to enter, “turn” decides who wins.

//flag[] is boolean array;
flag[0] = false
flag[1] = false
turn = 0  // or 1

// indicate intension
// If it is not my turn
// back off
// Finally, it is my turn
// re-indicate intension

P0:
flag[0] = true;
while (flag[1] == true) {
    if (turn != 0) {
        turn = 1;
        flag[1] = true;
    } else {
        flag[0] = false;
        while (turn != 0) {
            // busy wait
        }
    } // Finally, it is my turn
} // re-indicate intension

// critical section
...

P1:
flag[1] = true;
while (flag[0] == true) {
    if (turn != 1) {
        turn = 0;
        flag[0] = true;
    } else {
        flag[1] = false;
        while (turn != 1) {
            // busy wait
        }
    } // Finally, it is my turn
} // re-indicate intension

// critical section
...

// remainder section
// remainder section
Peterson’s algorithm – simplify Dekker’s

```c
#define FALSE 0
#define TRUE 1
#define N 2 /* number of processes */

int turn; /* whose turn is it? */
int interested[N]; /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other; /* number of the other process */

    other = 1 - process; /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process; /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

What if two processes execute “turn = process” almost the same time?

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}

Process 0: “turn = 0” // then Process 1: “turn = 1”
Process 0: while( turn == process && ...) // loop ends as “turn == 1”
It means whoever executes “turn = process” first wins
Lamport’s bakery algorithm – N processes

“A New Solution of Dijkstra's Concurrent Programming Problem”, Lamport 1974

```
lock (int pid) {
    // enter doorway
    choosing[pid] = 1;
    number[pid] = 1+max(number[0],...,number[N-1]);
    choosing[pid] =0;

    // enter bakery
    while(j<N) {
        if (j != pid) {
            while(choosing[j])
            while(number[j] &&
                ((number[j] < number[pid]) ||
                ((number[j]==number[pid]) && j<pid))
            ); // spin
        }
        j++;
    }
}

unlock (int pid) { number[pid] = 0; }
```

The current process is choosing its number
Process j’s choosing its number
The process that has the smallest number wins
Number 0 is reserved to indicate the exit of critical section
Hardware-assisted solutions

test_and_set(int* p) {
    int t = *p;
    *p = 1;
    return t;
}

//compare-and-swap
CAS(p, old, nvalue) {
    if(*p != old)
        return false;
    *p = nvalue;
    return false;
}

Load-link/store-conditional, or LL/SC // CAS-equivalent in RISC architectures

XCHG(int *x, int *p) {
    int t = *x;
    *x = *p;
    *p = t;
}

enter_region() {
    while(test_and_set(&lock) == 1);
}

leave_region() { lock = 0;}

enter_region() {
    a = 1;
    do { XCHG(&a, &lock);
    } while (a == 1);
}

leave_region() { lock = 0;}

enter_region() {
    a = 1;
    do { XCHG(&a, &lock);
    } while (a == 1);
}

leave_region() { lock = 0;}

CIS 5512 – Operating Systems 45
Outline

• IPC for passing data
  – Pipe, FIFO, Message Queue, Shared Memory
• Concepts
  – Race condition, critical section, and mutual exclusion
• Mutual exclusion based on busy waiting
  – Software-based solutions
  – Hardware-assisted solutions
• Synchronization without busy waiting
  – Kernel-assisted solutions: semaphore
  – Language-assisted solutions: monitor
Semaphore – avoids busy-waiting


• Previous solutions perform busy waiting
• Semaphore internals:
  – A counter to indicate the number of resources
  – A queue for processes waiting for the semaphore
• Semaphore needs operating system support
Semaphore – Two main APIs

```cpp
// Atomic
down(S) { // also called P operation
    if(S.num>0)
        S.num--;
    else
        put the current process in queue;
        the current process blocks;
}

// Atomic
up(S) { // also called V operation
    if(any process is in S’s wait queue)
        pop a process from the queue;
        resume this process;
    else
        S.num++;
}
```
Enforcing Mutex (mutual exclusion) using a binary semaphore

- If a semaphore’s counter value is restricted to 0 and 1, this semaphore is a **binary semaphore**
- A binary semaphore can be used to enforce mutual exclusion
  - The semaphore’s counter is initialize to 1
  - After a process X calls P(), all other process that call P() sleep in the wait queue of the semaphore
  - After X calls V(), one of the processes in the wait queue is waken up
Cons of Semaphore?

- When the competition for the resources is intense, suspending and resuming processes, (i.e., process switching) frequently is expensive.
- Assume a critical section contains $N$ instructions and a context switch takes $M$ instructions.
- If $N < M$, it actually saves CPU cycles to use busy waiting, compared to using Semaphore.
- That is why some systems use a hybrid method.
  - The process first spins for a while, and then
  - Blocks
### Semaphore and `pthread_mutex` in Linux

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sem_open()</code></td>
<td>Create or connect to a named semaphore (and increment the reference count), which can be conveniently shared inter-process</td>
</tr>
<tr>
<td><code>sem_unlink()</code></td>
<td>Remove the named semaphore once its reference count = 0</td>
</tr>
<tr>
<td><code>sem_close()</code></td>
<td>Decrement the reference count (<code>exit()</code> does this automatically)</td>
</tr>
<tr>
<td><code>sem_init()</code></td>
<td>Create an unnamed semaphore (usually a global variable)</td>
</tr>
<tr>
<td><code>sem_destroy()</code></td>
<td>Destroy an unnamed semaphore</td>
</tr>
<tr>
<td><code>sem_wait()</code></td>
<td>P operation</td>
</tr>
<tr>
<td><code>sem_post()</code></td>
<td>V operation</td>
</tr>
</tbody>
</table>
| `pthread_mutex_init()` | PTHREAD_PROCESS_SHARED: process-shared mutex  
Other APIs: *__destroy(), __lock(), __trylock(), __unlock() |
Monitor

- A programming language construct that supports controlled access to shared data; it encapsulates
  - Shared data
  - Procedures that operate on the shared data
  - Synchronization between concurrent processes that invoke those procedures (only one thread can execute any of the procedures at a time)

- A Monitor has a mutex lock and a queue
  - Processes has to acquire the lock before invoking a procedure in the monitor
  - If the lock has been acquired by a process, other requesting processes are put in the queue
(1) Shared data can only be accessed through procedures
(2) Only one thread can enter the monitor to invoke procedures at a time
(1) + (2) => It is guaranteed that all threads access the shared data in a mutual exclusive way
Monitor is easy to use and fool-proof, but less flexible than semaphores
Monitor in Java

• “Synchronized” non-static methods are to implement a Monitor object
  – A thread acquires the lock of an object to execute a synchronized method
  – If another thread tries to execute any of the synchronized methods, the thread blocks

• “synchronized” static method is to implement a Monitor class

• “synchronized block” is more flexible
What if a process has acquired a lock but cannot progress because some condition is not met?

• Consider the following problem:
  – A process acquired the lock and did a lot of computation, but then it finds that it cannot make progress as some condition (e.g., account >= 100) is not met

• Does the thread have to give up all the computation it has done? (In some cases, it may even have to rollback)

• Any better solution?
Condition Variable (CV)

• Condition variables provide a better solution
  – By using a CV, the thread can be suspended at the condition evaluation location, and resumes from where it is suspended when it is notified

• Each condition variable correspond to a wait queue, and there are three operations
  – Wait(c): when this is called, it is assumed that the calling thread has acquired the lock
    • The calling process is put in the wait queue of the CV
    • Release the monitor lock (so somebody else can get in)
  – Signal(c)
    • Wake up one process in the wait queue, and the process goes to wait in the queue of the monitor lock
    • Once it acquires the monitor lock, it can resume from where it was suspended
  – Broadcast(c)
    • Wake up all processes in the wait queue
## Condition Variable

<table>
<thead>
<tr>
<th>Operations</th>
<th>Posix</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait(c)</td>
<td><code>pthread_cond_wait (cond, mutex)</code></td>
<td><code>wait()</code></td>
</tr>
<tr>
<td>Signal(c)</td>
<td><code>pthread_cond_signal(cond)</code></td>
<td><code>notify()</code></td>
</tr>
<tr>
<td>Broadcast(c)</td>
<td><code>pthread_cond_broadcast(cond)</code></td>
<td><code>notifyAll()</code></td>
</tr>
</tbody>
</table>
How to use a Condition Variable

1. Always hold the lock while performing CV operations
2. Always put the wait operation in a loop
   - In order to check whether the condition is true
   - Note the the `signal(c)` only hints the conditions changes; it does not imply that the condition is definite true. Moreover, after signaling, some other processes may affect the condition

```c
Strange_withdraw(x) {
  ...
  pthread_mutex_lock(&m);
  while(account < x) // “if” will not work
      pthread_cond_wait(&c, &m);
  account -= x;
  pthread_mutex_unlock(&m);
}

Deposit(y){
  pthread_mutex_lock(&m);
  account += y;
  pthread_cond_signal(&c);
  pthread_mutex_unlock(&m);
}
```

```c
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```
“Always hold the lock while performing CV operations”, why?

- The correct use of “Wait(c)” requires that the caller holds the lock, since the semantics of this call implies the release of the lock.
- If the caller of “signal(c)” does not hold the lock, there may be race conditions.
  - Consider `w.1 -> s.1 -> s.2 -> w.2` => signal lost

```c
Strange_withdraw(x) {
    ...
    pthread_mutex_lock(&m);
    while(account < x) // w.1
        pthread_cond_wait(&c, &m); // w.2
    account -= x;
    pthread_mutex_unlock(&m);
}

Deposit(y){
    // pthread_mutex_lock(&m);
    account += y; // s.1
    pthread_cond_signal(&c); // s.2
    // pthread_mutex_unlock(&m);
}
```
Relations between Condition Variable & Monitor

• A Monitor may contain zero or more CVs
  – Very often, procedures in Monitor rely on CVs to implement complex synchronization
  – Recall that a CV has to be used with a lock; a Monitor can provide the lock, so you do not have to explicitly use a lock for employing a CV in a Monitor

• The use of CVs is not limited to Monitors
  – E.g., Pthread library provides CVs but not Monitors
Condition variable VS Semaphore

- A Semaphore has a counter and a wait queue, while a Condition Variable only has a wait queue
  - Plus, you need to initialize the counter when using a Semaphore, and the initialization value means the number of resources. A Condition Variable has no notion of “the number of resources”

- Condition Variables allow broadcast() operation, while Semaphores do not
Issues with locks (mutex and semaphore)

- You may introduce **deadlock**
- **Nobody can make progress** sometimes (when the lock owner is scheduled out)
- **Priority inversion**: a high priority thread is waiting for the lock occupied by a low priority thread, which is then preempted by a medium priority thread
- **Solution**: **non-blocking** programming / data structures (but they are very complex and, very often, slow)
Big picture of synchronization primitives

Sync. Primitives

Busy-waiting

Software solutions (Dekker’, Bakery, etc.)

Hardware-assisted solutions (based on atomic read-modify-write instructions)

Semaphore: it contains an internal counter indicating the number of resources.

Binary Semaphore is a special semaphore, whose counter value can only be 0 or 1; it can be used as a mutex

Monitor: a high-level synchronization construct; shared data + procedures + lock

Condition variable: guarded by a lock to wait on some event

Blocking

CIS 5512 – Operating Systems
Take away…

• IPC
  – Pipes, FIFO, message queues, and shared memory
• Concepts
  – Race condition, critical section, mutual exclusion
• Pure software based locks
  – Dekker’s, Peterson’s, Bakery algorithms
• Atomic read-modify-write (RMW) instructions
  – Test-and-set, xchg, compare-and-swap
  – Mutex based on (RMW)
• Semaphores and binary semaphores
• Monitors and Conditional variables
• Lock-free concurrent computing