A modern computer system

CPU

Disk controller

Disks

Mouse

Keyboard

USB controller

Printer

Graphics adapter

Memory

OS

device driver

device driver

App

App

App
HW-OS interface

CPU

device controller

memory

OS

device driver

application
OS-App interface
From program code to a process

**Program code**  
Written in a high-level language

**Compilation**

**Executable image**  
Sequence of machine instructions and data

**Process**  
Instance of a program, ready to be executed by CPU.

The OS copies the executable into memory, and reserves memory for stack and heap.

```
if (x>0)  
y:=1  
else  
y:=2
```

```
00011101  
11110000  
10101000  
11110010
```

```
<table>
<thead>
<tr>
<th>instructions</th>
<th>data</th>
<th>heap</th>
<th>stack</th>
</tr>
</thead>
</table>
```
From a process to threads

single-threaded process

multi-threaded process
Scheduling Algorithms

FIFO
- Simplicity
- Lateness
- Low overhead
- Turnaround time
- Response time
- Starvation freedom

SJF

RR
A Multi-level System

I/O bound jobs

CPU bound jobs

priority

timeslice


Need for synchronization

- For a multithreaded program to be correct, some restrictions on accessing shared data by threads should be satisfied.
- Threads’ access to shared resources should be coordinated.
- Threads should coordinate on their own their access to shared data.
- All threads should still be able to make progress!
Share Counting with lock

\[ \text{bills\_counter} = 0 \]
\[ \text{lock = released} \]

- **Thread A**
  ```
  \text{while} (\text{machine\_A\_has\_bills}) \\
  \text{acquire (lock)} \\
  r1 = \text{bills\_counter} \\
  r1 = r1 + 1 \\
  \text{bills\_counter} = r1 \\
  \text{release (lock)}
  ```

- **Thread B**
  ```
  \text{while} (\text{machine\_B\_has\_bills}) \\
  \text{acquire (lock)} \\
  r2 = \text{bills\_counter} \\
  r2 = r2 + 1 \\
  \text{bills\_counter} = r2 \\
  \text{release (lock)}
  ```
Producer-Consumer Problem

Shared data: buffer, “In”, “Out”

Shared Semaphores: mutex, empty, full;

mutex = 1; /* for mutual exclusion*/
empty = N; /* number empty buf entries */
full = 0; /* number full buf entries */

**Producer**
do {
    P(empty);
P(mutex);
    //produce item
    //update “In”
    V(mutex);
    V(full);
} while (true);

**Consumer**
do {
    P(full);
P(mutex);
    //consume item
    //update “Out”
    V(mutex);
    V(empty);
} while (true);
Readers-Writers Problem

mutex = Semaphore(1)
wrt = Semaphore(1)
readcount = 0;

**Writer**
do{
  P(wrt);
  /*writing is performed*/
  V(wrt);
}while(true)

**Reader**
do{
  P(mutex);
  readcount++;
  if (readcount == 1)
    P(wrt);
  V(mutex);
  /*reading is performed*/
  P(mutex);
  readcount--;
  if (readcount == 0)
    V(wrt);
  V(mutex);
}while(true)
Monitor

• A data abstraction mechanism, which consists of:
  – state and
  – procedures.
• The state is modeled by shared variables.
• The procedures are the only means by which the state is manipulated.
• Mutual exclusion: only one thread can execute a monitor procedure at any time.

```
Monitor monitor_name
{

  // shared variable declarations

  procedure P1(. . ) {
    . . .
  }

  procedure PN(. . ){
    . .
  }

  initialization_code(. . .){ . . .}
}
```
A Simple Monitor

Monitor EventTracker {
    int numbburgers = 0;
    condition hungrycustomer;

    void customerenter() {
        while (numburgers == 0)
            hungrycustomer.wait()
        numbburgers -= 1
    }

    void produceburger() {
        ++numburgers;
        hungrycustomer.signal();
    }
}
Synchronization: abstraction layers

- Locks (acquire, release), semaphores (Init, P, V), condition variables (wait, signal)
- Spinlocks, queuing locks
- TestAndSet, disable interrupts
Synchronization primitives

- All can encode any predicate on shared data.
- Each primitive can be used to implement another primitive.

<table>
<thead>
<tr>
<th>Locks</th>
<th>Semaphores</th>
<th>Condition Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>(acquire, release)</td>
<td>(init, P, V)</td>
<td>(wait, signal)</td>
</tr>
</tbody>
</table>

Decreasing programming effort to encode predicates
Deadlock

semaphore: mutex1 = 1  /* protects file */
            mutex2 = 1  /* protects printer */

Process A code:
{
    /* initial compute */
    P(mutex1)
P(mutex2)
P(mutex2)
/* use file & printer*/
V(mutex2)
V(mutex1)
}

Process B code:
{
    /* initial compute */
P(mutex2)
P(mutex1)
    /* use file & printer */
V(mutex1)
V(mutex2)
}
Four Conditions for Deadlock

- Mutual Exclusion
- Hold and wait
- No preemption
- Circular wait
Banker’s Algorithm

For a request $R$ of additional resources issued by process $P$, which is the next process scheduled to run:

1. If $R$ does not exceed $P$’s maximum claim, go to 2. Otherwise, error.

2. If $R$ does not exceed the available resources, go to 3. Otherwise, $P$ should wait.

3. Pretend that $R$ is granted to $P$.
   Update the state of the system.
   If the state is safe, then give requested resources to $P$. Otherwise, $P$ should wait and the old state is restored.
Memory: allocation strategy

• Should processes have contiguous space of physical addresses in memory?

• Is memory partitioned into fixed- or variable-sized segments?
  – If variable-sized segments, which allocation algorithm is used?
    • First fit: allocate first hole that is big enough.
    • Best fit: allocate the smallest hole that is big enough.
    • Worst fit: allocate the largest hole.
Address translation

• The CPU understands virtual addresses.
• The memory unit understands physical addresses.
• The OS and specialized hardware are responsible for translating virtual addressed into physical addresses.
• The translation mechanism gives protection.
Paging

• Divide physical memory into frames:
  – Fixed-sized blocks.
  – Size is power of 2, between 512 bytes and 8,192 bytes.

• Divide virtual memory into pages.
  – Same size as frames.

• Page table translates virtual to physical addresses.
Address Translation Scheme
Hierarchical Paging
Page Replacement Algorithms

- **FIFO**: the page brought in earliest is evicted.
- **OPT**: evict page that will not be used for the longest period of time.
- **LRU**: evict page that has not been used the longest.
- **MRU**: evict the most recently used page.
- **LFU**: evict least frequently used page.
Coming up...

- Next lecture: File System Interface
- Exam2: tomorrow last 30mis