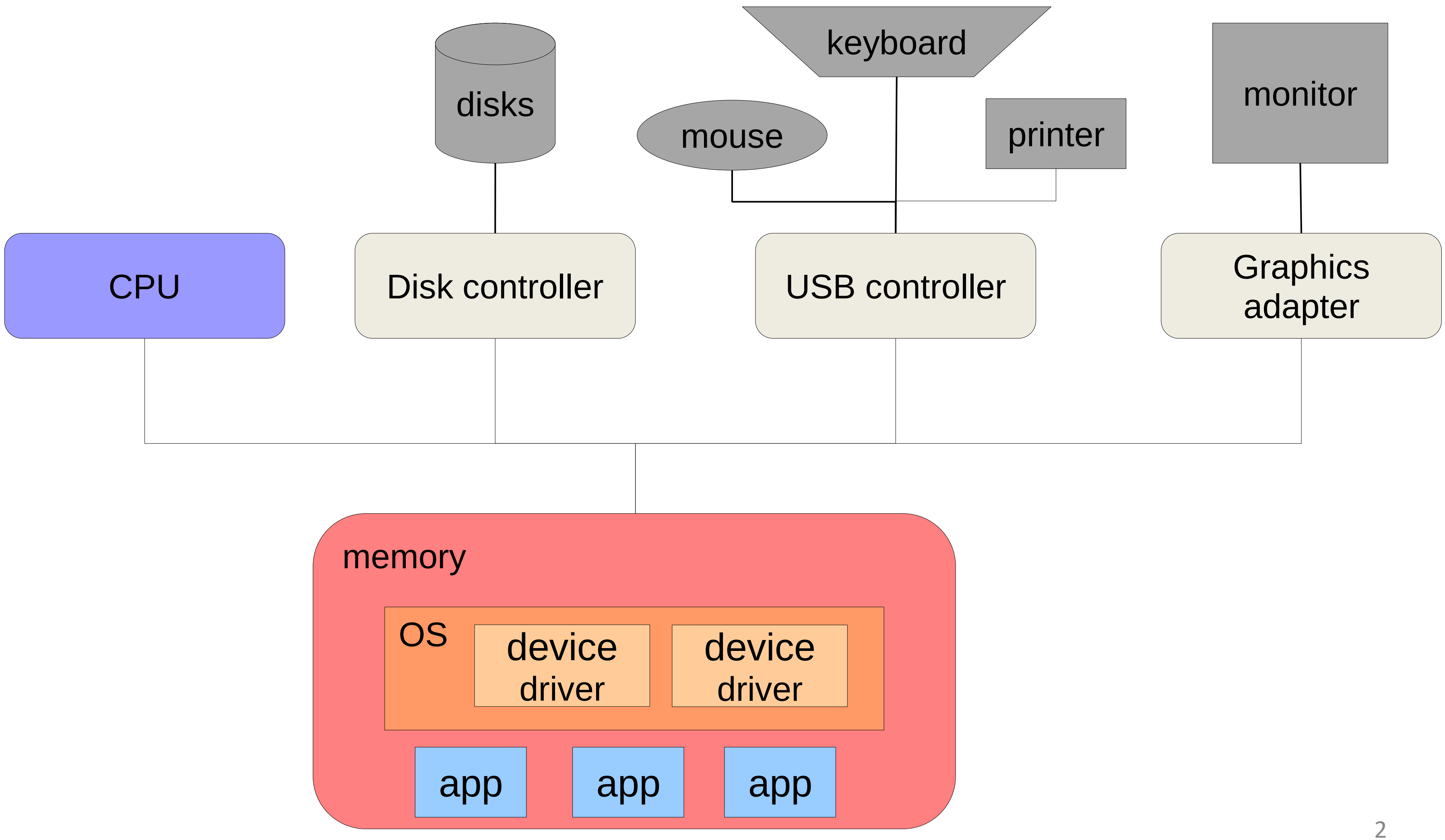
CS 4410
Operating Systems

Review 1

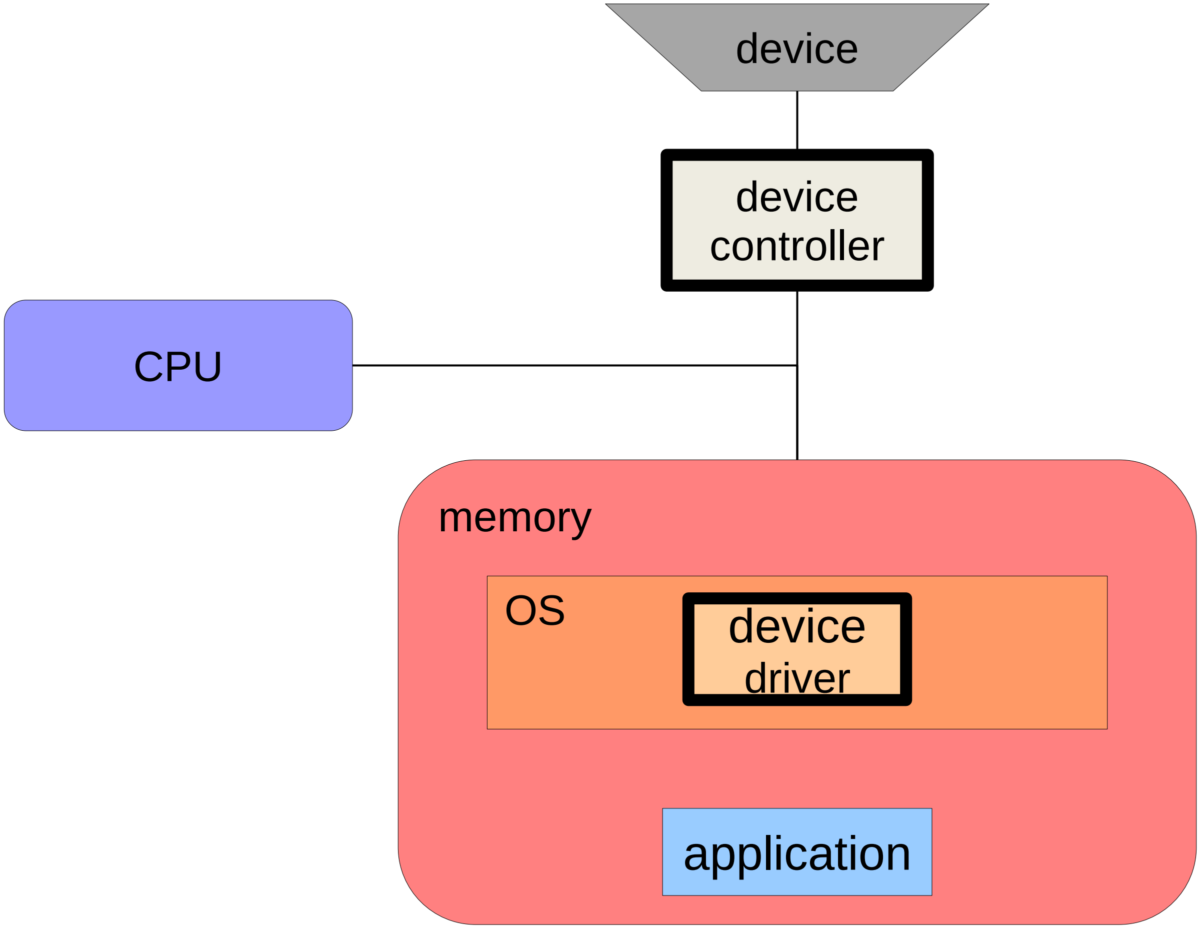
Summer 2016

Cornell University

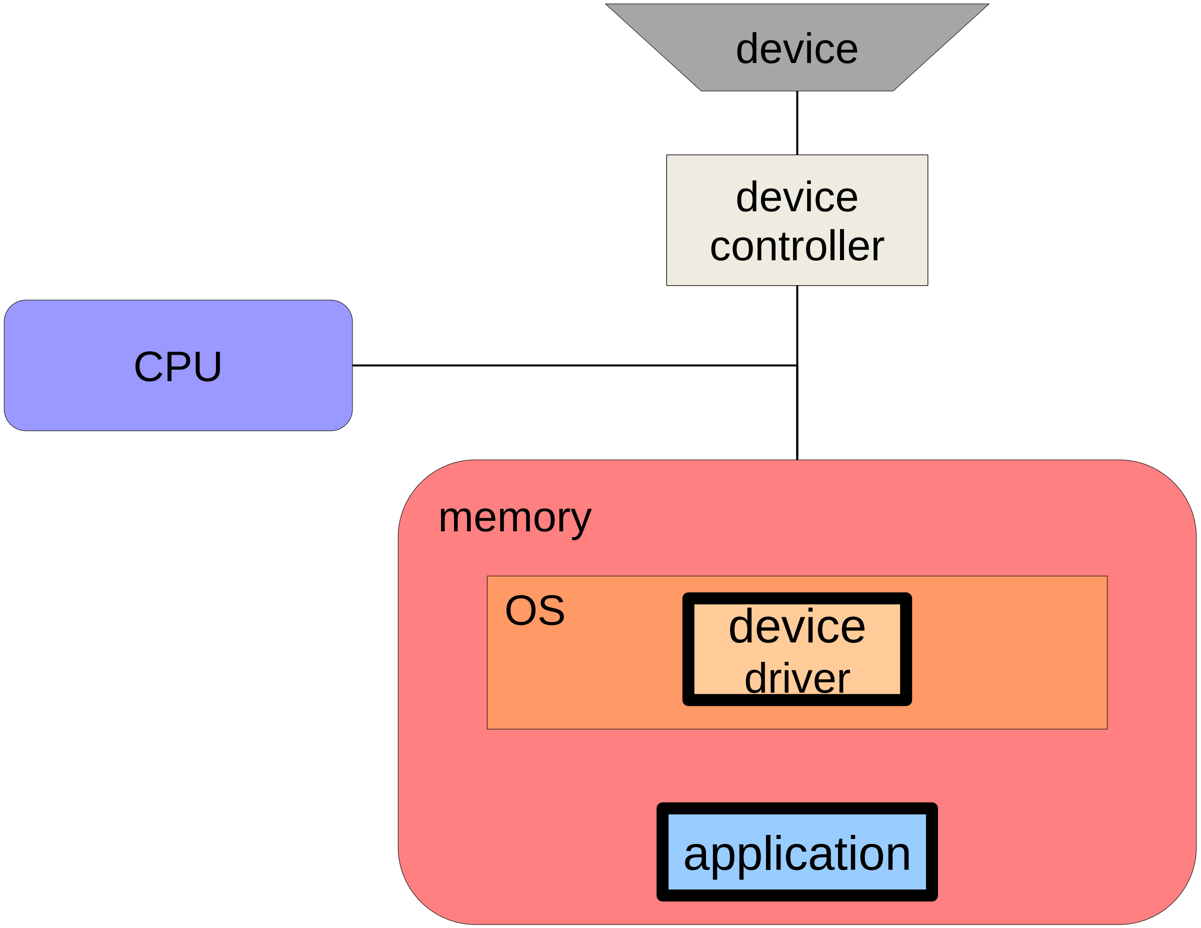
A modern computer system



HW-OS interface

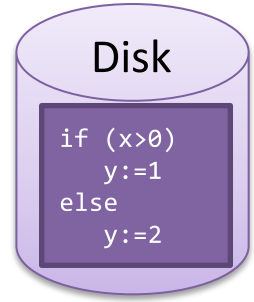


OS-App interface



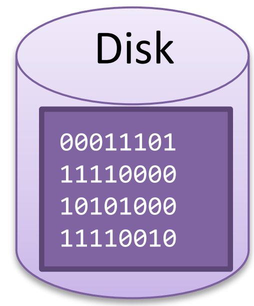
From program code to a process

Program code — Written in a high-level language



↓ Compilation

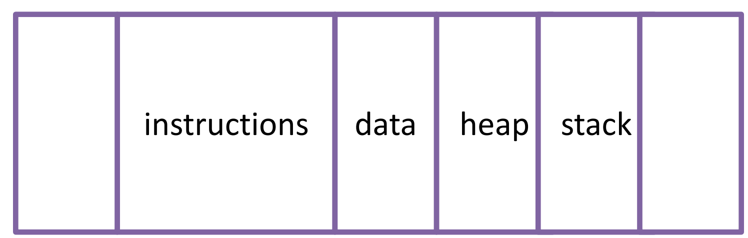
Executable image — Sequence of machine instructions and data



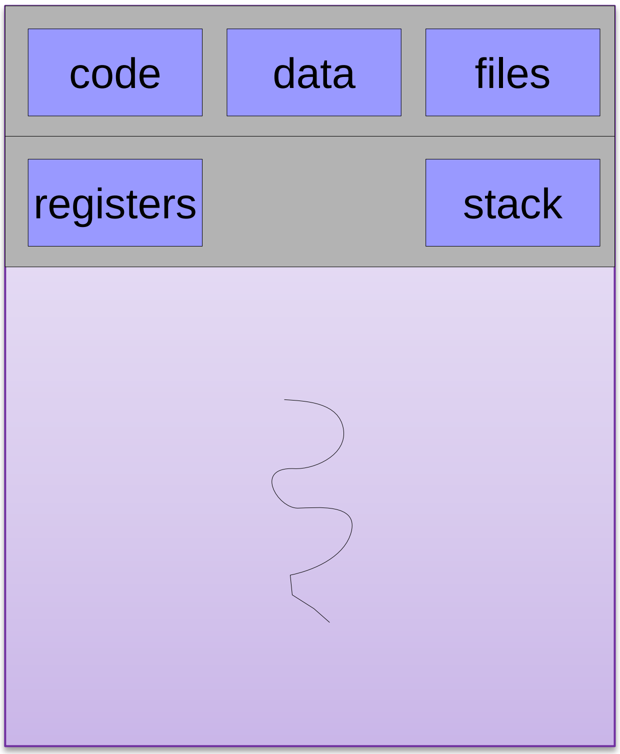
↓ The OS copies the executable into memory, and reserves memory for stack and heap.

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

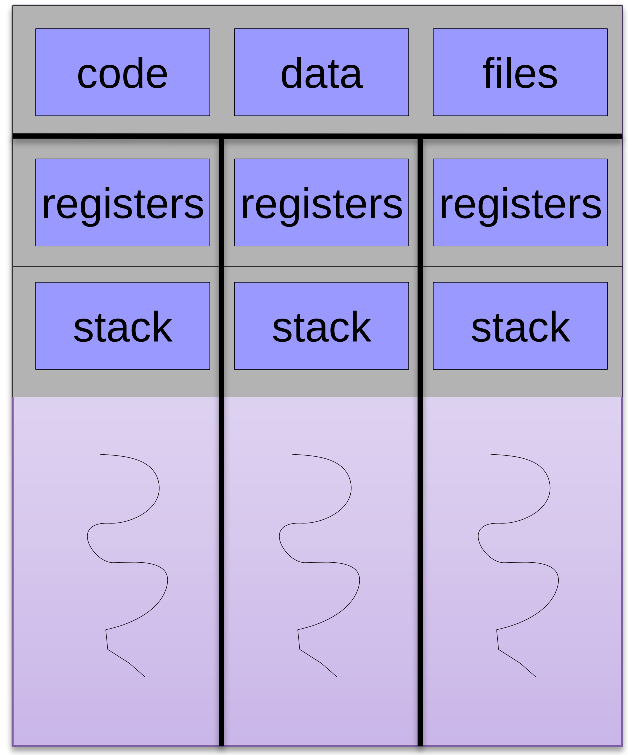
Memory



From a process to threads

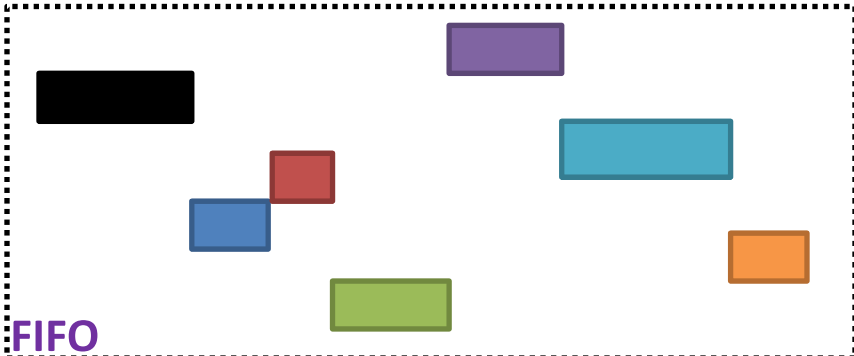


single-threaded process



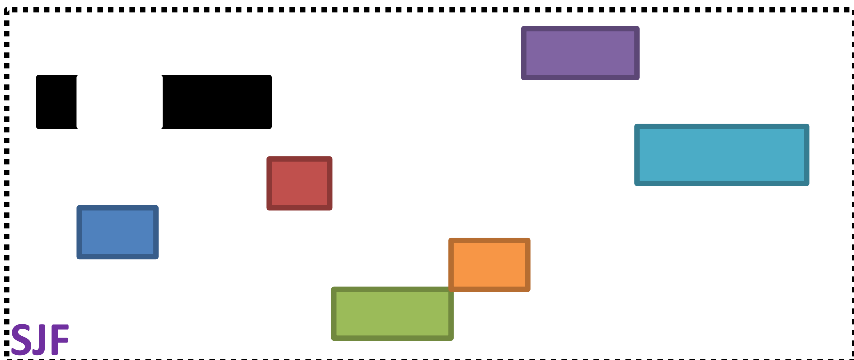
multi-threaded process

Scheduling Algorithms



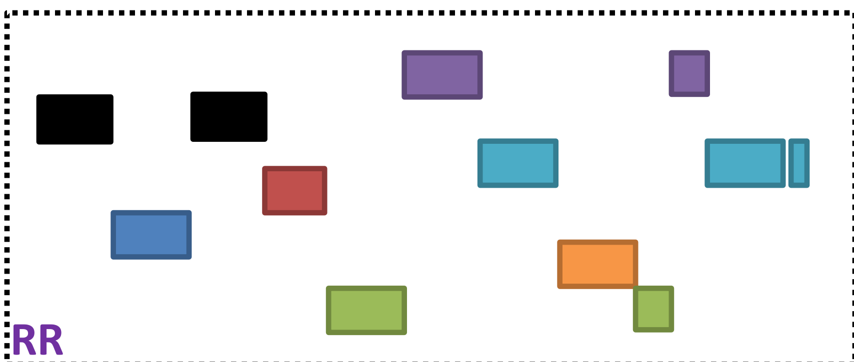
Simplicity

Low overhead



Lateness

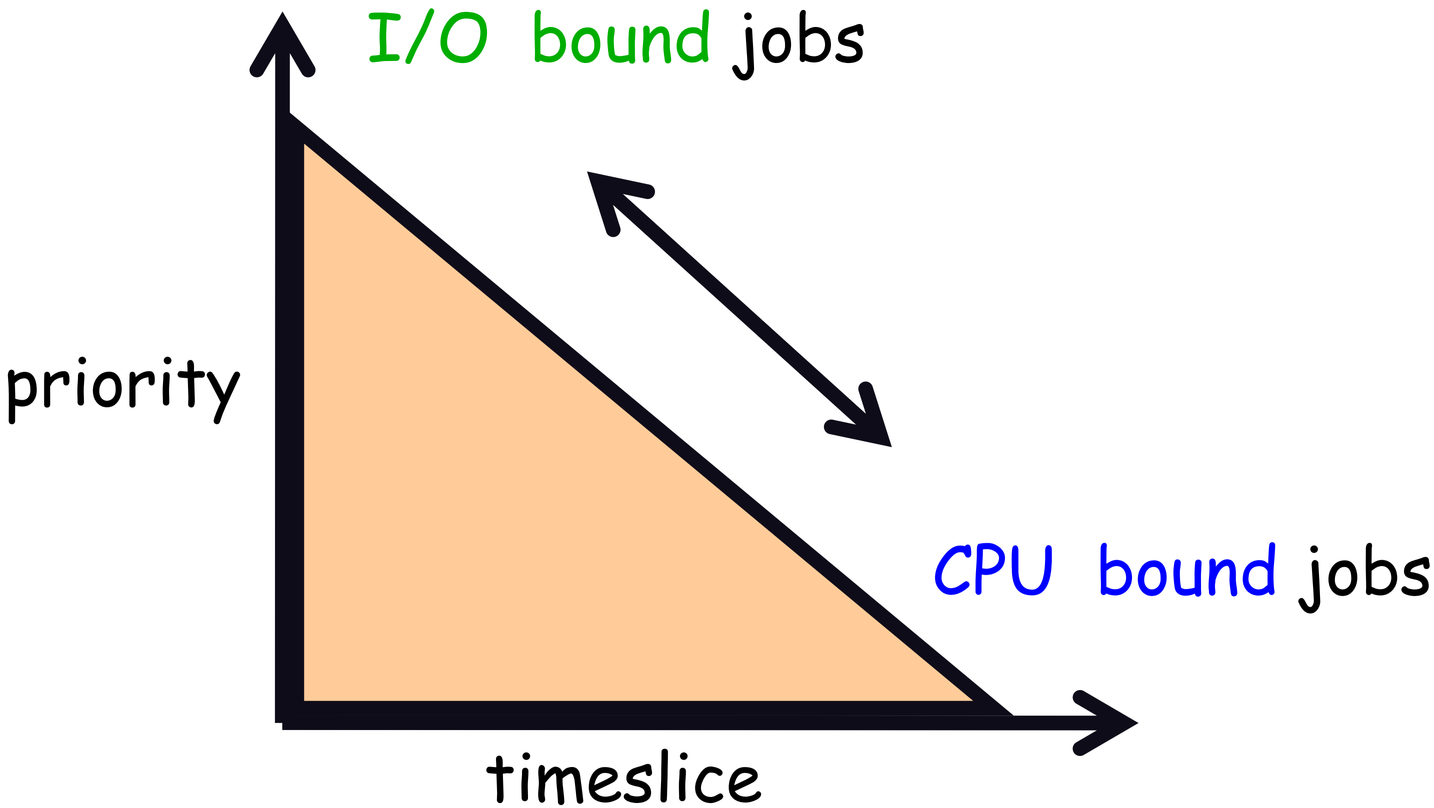
Turnaround time



Response time

Starvation freedom

A Multi-level System



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

bills_counter = 0
lock = released

- Thread A

while (machine_A_has_bills)

    acquire (lock)

        r1 = bills_counter

        r1 = r1 +1

        bills_counter = r1

    release (lock)

- Thread B

while (machine_B_has_bills)

    acquire (lock)

        r2 = bills_counter

        r2 = r2 +1

        bills_counter = r2

    release (lock)

Critical Section

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 */
```

<u>Producer</u>

```
do {
    P(empty);
    P(mutex);
      //produce item
      //update "In"
    V(mutex);
    V(full);
} while (true);
```

<u>Consumer</u>

```
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 (reardcount == 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(. . . ) {  . . . }
}
```

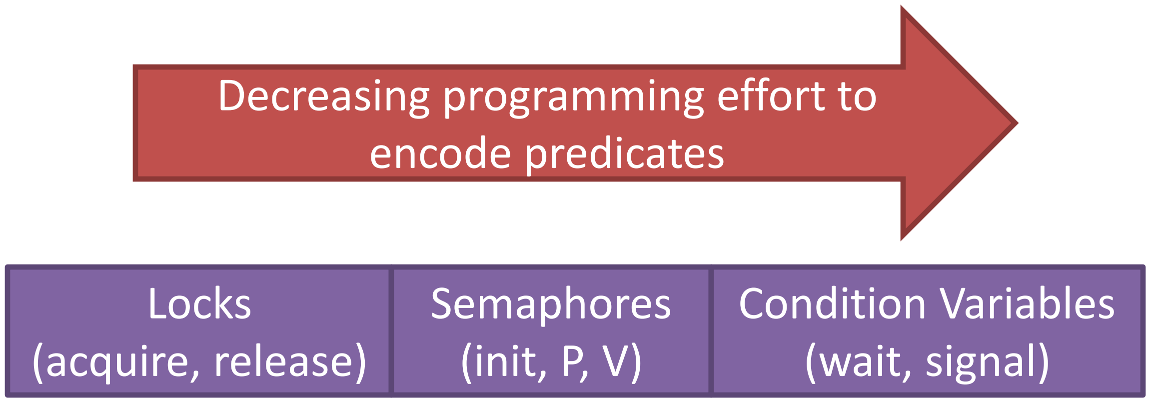
A Simple Monitor

```
Monitor EventTracker {
  int numburgers = 0;
  condition hungrycustomer;

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

Deadlock

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

Process A code:

```
{
    /* initial compute */

   P(mutex1)
   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.

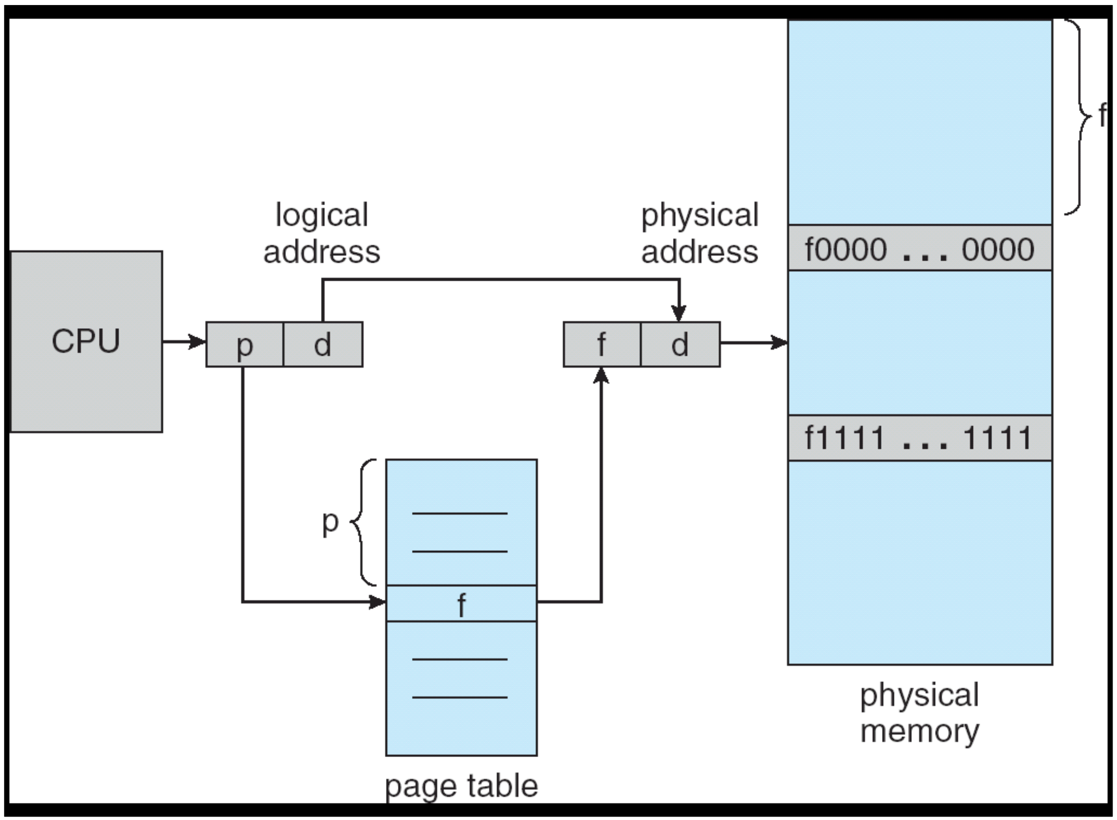
Address translation

- 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.

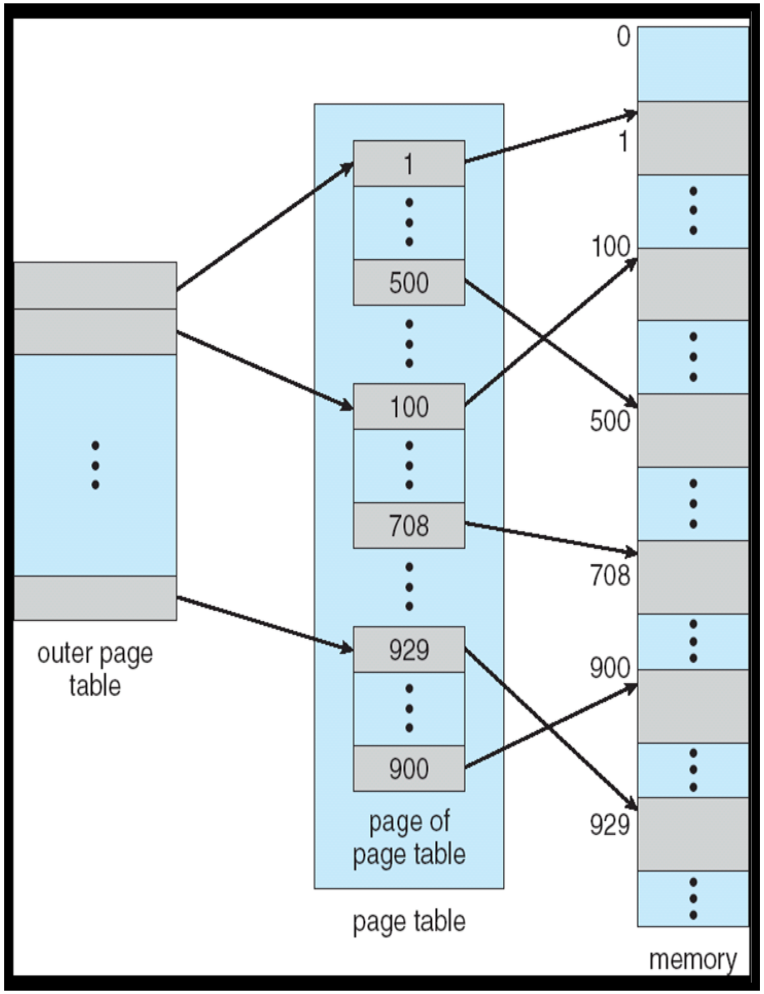
Paging

- 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 it earliest is evicted
- **OPT**: evict page that will not be used for 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