Classic Process Synchronization Problems

• A solution to a process synchronization problem should meet three important criteria:
  – Correctness
  – Maximum concurrency
  – No busy waits
• Some classic problems:
  – Producers-Consumers with Bounded Buffers
  – Readers and Writers
  – Dining Philosophers

Producers-Consumers with Bounded Buffers

• A solution must satisfy the following:
  1. A producer must not overwrite a full buffer
  2. A consumer must not consume an empty buffer
  3. Producers and consumers must access buffers in a mutually exclusive manner
  4. (Optional) Information must be consumed in the same order in which it is put into the buffers (FIFO)
Producers-Consumers with Bounded Buffers (continued)

Fig. 6.14 An outline for producers–consumers using critical sections.

- Suffers from two problems:
  – Poor concurrency and busy waits

Fig. 6.15 An improved outline for a single buffer producers–consumers system using signaling.
Producers-Consumers with Bounded Buffers (continued)

**Readers and Writers**

- A solution must satisfy the following:
  1. Many readers can perform reading concurrently
  2. Reading is prohibited while a writer is writing
  3. Only one writer can perform writing at any time
  4. (optional) A reader has a nonpreemptive priority over writers
     – Called readers preferred readers–writers system

---

**Figure 6.16** Indivisible operations for the producers–consumers problem.

**Figure 6.17** Readers and writers in a banking system.
Readers and Writers (continued)

Parbegin
repeat
  If a writer is writing
  then
    { wait }
    { read }
  If no other readers reading
  then
    if writer(s) waiting
    then
      activate one waiting writer;
    forever,
  end.
Write(s)

Figure 6.18 An outline for a readers-writers system.

Dining Philosophers

Figure 6.19 Dining philosophers.

- Design processes to represent the philosophers so that each philosopher can eat when hungry and none dies of hunger
Dining Philosophers (continued)

```plaintext
repeat
  if left fork is not available
    then
      block (P);
      lift left fork;
      if right fork is not available
        then
          block (P);
          lift right fork;
          ( eat )
          put down both forks
      if left neighbor is waiting for his right fork
        then
          activate (left neighbor);
      if right neighbor is waiting for his left fork
        then
          activate (right neighbor);
          ( think )
    forever
```

Figure 6.20 Outline of a philosopher process $P_i$.

- Prone to deadlocks and race conditions
- Avoid deadlock: If right fork is unavailable, release left fork, retry later. But this suffers from livelocks!

Dining Philosophers (continued)

```plaintext
var successful : boolean;
repeat
  successful := false;
  while (not successful)
    if both forks are available then
      lift the forks one at a time;
      successful := true;
    if successful = false
      then
        block (P);
        ( eat )
        put down both forks;
    if left neighbor is waiting for his right fork
      then
        activate (left neighbor);
    if right neighbor is waiting for his left fork
      then
        activate (right neighbor);
        ( think )
  forever
```

Figure 6.21 An improved outline of a philosopher process.

- Problem: loop causes a busy wait condition