PROCESS SYNCHRONIZATION

Module Number 2. Section 9b
COP4600 – Operating Systems
Richard Newman

OUTLINE

- Need for process synchronization – race conditions
  - Critical regions
- Requirements for synchronization solutions
  - Safety
  - Liveness – deadlock, starvation, livelock
- Shared Memory - Spin locks
- Semaphores
- Barrier Synchronization
- Monitors
- Message-passing
  - Blocking receive
  - Rendezvous
- Shadow copies

RACE CONDITIONS

Process

```
store(my_file,slot[in]);
in++;
```

PrintServ

```
while (out < in)
  print(slot[out]);
  out++;
```

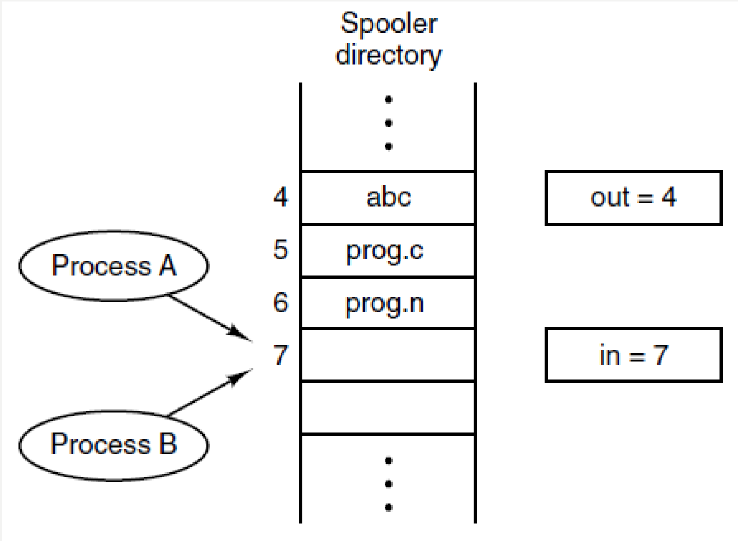
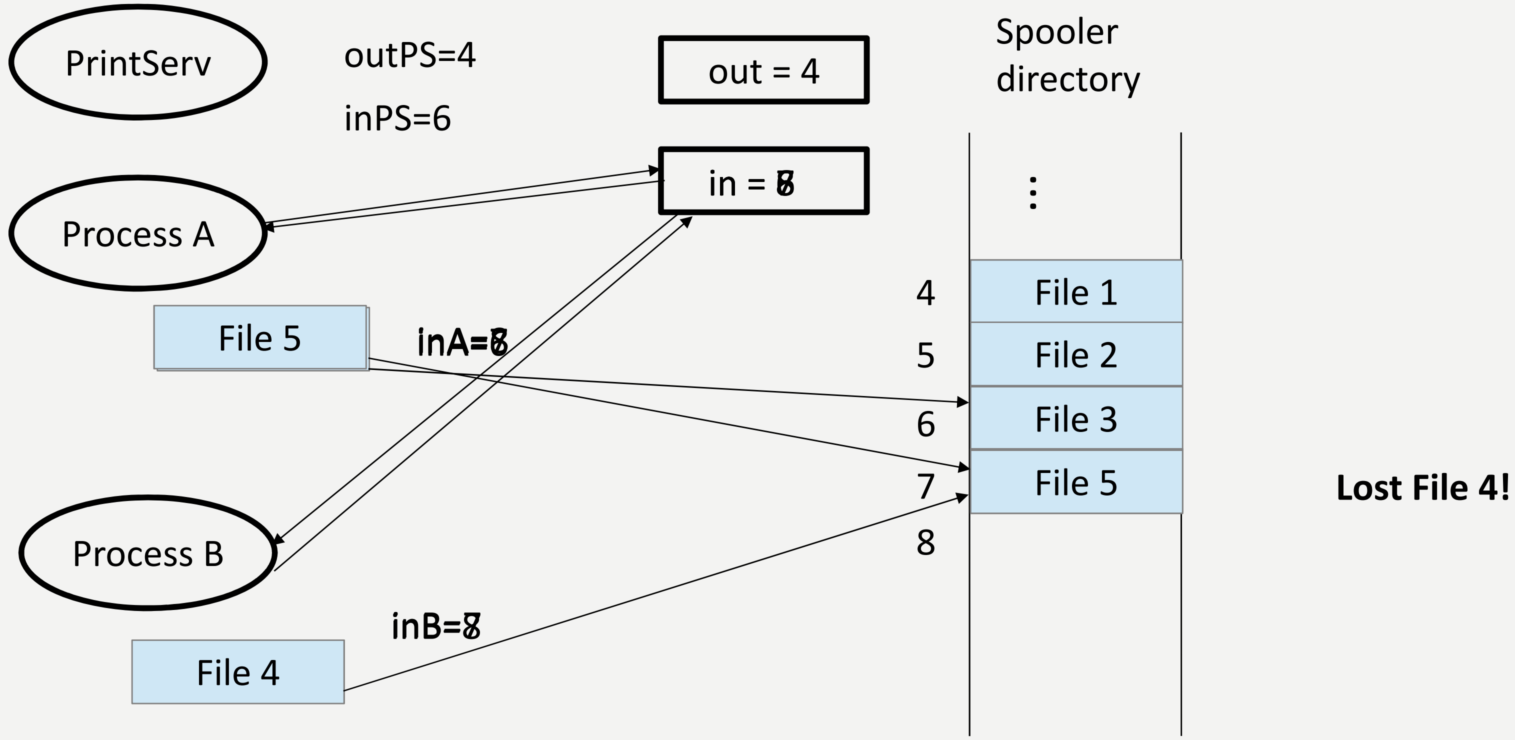


Figure 2-21. Two processes want to access shared memory at the same time.

RACE CONDITIONS



RACE CONDITION EXAMPLE

Load–Store atomicity:

Loads and stores occur atomically

X = X+1 translates to machine instructions:

Load X location contents into register

increment register

store register in X location

Lost Update:

P1 runs and loads X;

P2 runs and loads X, increments register, stores into X;

P1 runs again, increments register, stores into X

RACE CONDITION EXAMPLE (2)

P1

Load A, X

Incr A

Sto A, X

P2

Load A, X

Incr A

Sto A, X

“critical regions”

If initial value of X was 0, final value of X is 1, not 2.

Want to prevent other process from accessing X at same time!

CRITICAL REGIONS

Requirements to avoid race conditions:

1. No two processes may be simultaneously inside their critical regions. (Safety)
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes. (Liveness)
4. No process should have to wait forever to enter its critical region. (Liveness)

SYNCHONIZATION CRITERIA

Safety – what MUST NOT happen to avoid disaster

e.g., No two processes may be simultaneously inside their critical regions

Liveness – what MUST happen to keep everyone happy

e.g., No process running outside its critical region may block other processes

e.g., No process should have to wait forever to enter its critical region

The exact nature of these depend on system, but generally want to avoid *deadlock* and *starvation*

Deadlock = set of processes all wait on each other – none proceed

Starvation = some process waits indefinitely

CLASSIC SYNCHRONIZATION PROBLEMS

Critical Region Problem: Mutual exclusion – only one process can be in Critical Region at a time

Producer-Consumer Problem: Producers produce items and write them into buffers; Consumers remove items from buffers and consume them. A buffer can only be accessed by one process at a time.

Bounded Buffer Problem: P-C with finite # buffers

Readers-Writers: Any number of readers can read from a database simultaneously, but writers must access it solo

Dining Philosophers: Philosophers arranged in a circle share a fork between each pair of neighbors. Both forks are needed for a philosopher to eat, and only one philosopher can use a fork at a time.

SYNCHRONIZATION CODE- STARTUP

Generic Code for a starting a System of Processes:

```
main () {
    init_vars()        /* initialize shared variables */
    for (i=0; i<NUM_PROCS; i++) {
        StartProc(i)          /* fork and run process i */
        }
    for (i=0; i<NUM_PROCS; i++) {
         WaitProc(i)          /* wait for process i to terminate */
         }
   }
```

Processes may all be the same, or they may be of various types (depending on nature of system), or they may do one thing sometimes and something else other times;

BUT they all execute entry and exit codes for critical sections

GENERIC SYNCHRONIZATION CODE

```
Process (i) {
    while (TRUE) {
        PreOther(i)      /* arbitrarily long time outside CR */
        Entry(i)      /* code to make sure it's OK to enter */
          Critical_Region(i)         /* risky business */
        Exit(i)              /* code to let others know I'm done */
        PostOther(i)          /* arbitrarily long time outside CR */
    }
}
```

Key point: Often want to allow multiple processes to do dangerous stuff at the same time, but the entry and exit code prevents bad stuff from happening – i.e., not always critical region problem

SYNCHRONIZATION MECHANISMS

Basic flavors of synchronization mechanisms:

Busy waiting (spin locks) – S/W or H/W based

See these in Critical Region lesson

Semaphores (scheduling based) – OS support

Barrier Synchronization – OS support

Monitors – language support

Message-based – blocking receive, OS support

... others as well ...

SPIN LOCKS

- S/W or H/W based
- Entry code consists of testing a shared variable or memory location (the lock)
- Tight loop until lock is available
  - Uses CPU until TRO or lock available
  - Usually only a few instructions
  - Set lock when it is available
- Exit code consists of resetting lock
  - Usually a single instruction

SEMAPHORES

Semaphores are a *special type* of shared memory:

1 – A semaphore S can only be declared and initialized, or accessed by Up(S) or Down(S);

NO direct access

2 – When Down(S) is called, if S > 0, S-- and continue; otherwise block the caller and put on S's queue Q

3 – When Up(S) is called, if Q empty, S++ and continue; otherwise remove a process P from Q and unblock P

4 – Up() never blocks

5 – A process blocked on Down() does not use CPU

Scheduling-based synchronization

SEMAPHORES TYPES AND USES

Binary Semaphores: take only values 0 or 1

Uses:

- Initialized to 1, used for mutual exclusion (a process does Down(mutex); CS; Up(mutex))
- Initialized to 0, used for synchronization (one process does Down(S) to wait for another process to reach a point in its code where it does Up(S))

Multivalued Semaphores: take integer values >= 0

Uses:

- Initialized to N, where N is the number of instances of a shared resource
- Up(S) to release an instance, Down(S) to take one

BINARY SEMAPHORE FOR MUTUAL EXCLUSION

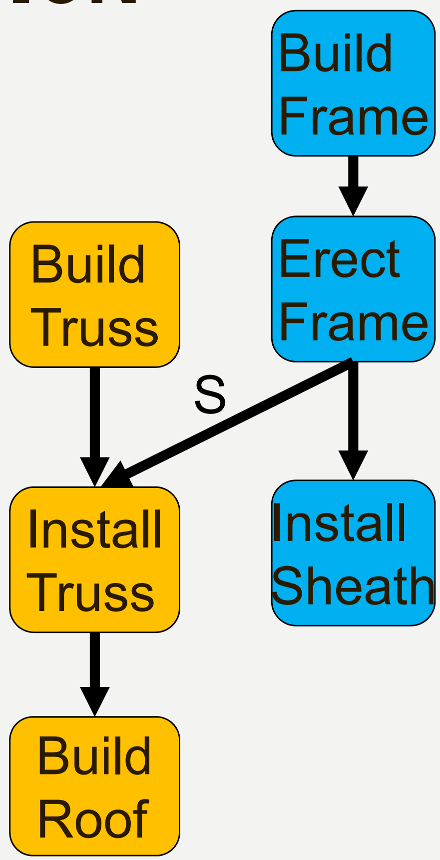
```
Semaphore S=1;

Process (int  i) {
    while (TRUE) {
    Other(i);       /* arbitrarily long time outside CR */
    Down(S);        /* code to make sure it's OK to enter */
        Critical_Region(i)      /* risky business */
    Up(S)                   /* code to let others know I'm done */
    }
  }
```

BINARY SEMAPHORE FOR PROCESS SYNCHRONIZATION

```
Semaphore S=0;

Process A {
        FirstOfA();
        Down(S);
        LastOfA();
    }
Process B {
        FirstOfB();
        Up(S);
        LastOfB();
  }
```



Precedence Graph

```
Semaphore S=0;

Process Roof {
        BuildTrusses();
        Down(S);
        InstallTrusses();
        BuildRoof();
    }
Process Walls {
        ErectWallFrames();
        Up(S);
        InstallSheathing();
  }
```

PRECENDENCE GRAPH

- Directed Acyclic Graph (DAG)
- Nodes = tasks
- Arcs = precedence relation
  - If A → B, then A must be completed before B can begin
- Unrelated tasks can be done in parallel
- Precedence can be enforced by
  - Normal control flow (A and B in same process)
  - Synchronization semaphores

BARRIERS

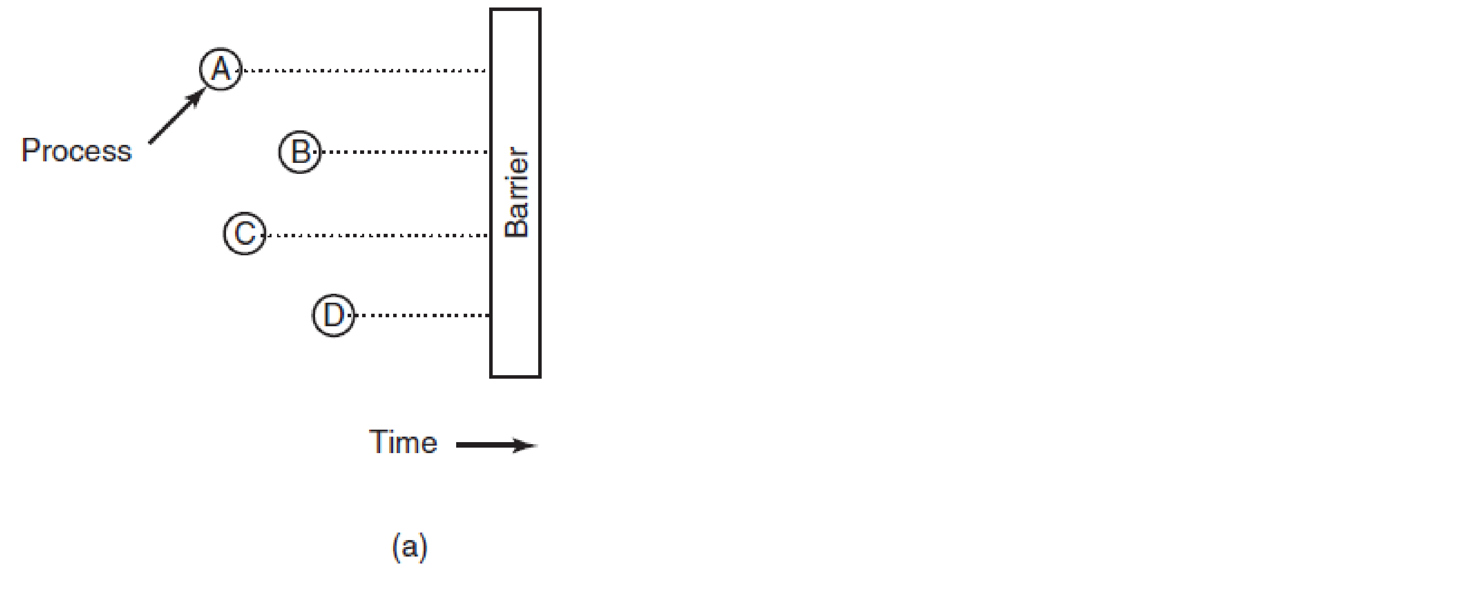


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier.
(b) All processes but one blocked at the barrier.
(c) When the last process arrives, all of them are let through.

MONITORS

Monitors are a *language construct*

– With spin locks and semaphores, programmer must remember to obtain lock/semaphore, and to release the same lock/semaphore when done: this is prone to errors!

– Monitor acts like an Abstract Data Type or Object – only access internal state through public methods, called entry procedures

– Monitor guarantees that at most one process at a time is executing in any entry procedure of that monitor

– What if a process can't continue while in entry proc?

CONDITION VARIABLES

- What if a process can't continue while in entry proc?
- Condition “variables” local to monitor allow a process to block until another process signals that condition
- Unlike semaphores, condition vars have NO memory!
- If no process is waiting on the condition, signaling the condition variable *has NO effect at all*!
- A process that waits on a condition variable can be awakened and resume execution in entry procedure
- So what about the process that woke the waiting process up?
  - May force it to wait (on a stack), or
  - May require it to leave entry procedure

MONITOR SYNTAX

```
monitor <name>
    <shared var decls>

    <entry procedures>

    <init code block>
end monitor;
```

```
monitor foo
    integer count;
    condition cond1, cond2;

    procedure bar(int j)
    begin
        if (not ready) wait(cond1)
        …
        signal(cond2);
    end

    count = 0;
end monitor;
```

MESSAGE-PASSING

- Message-passing primitives
  - Send – to destination
  - Receive – from origin or from any
- Blocking – can be used for synchronization!
  - Receive normally blocks if no message
  - Send is often non-blocking
  - Rendezvous – both send and receive block
- Storage
  - Where is undelivered message stored?
  - What is capacity?

SHADOW COPIES

- When performing an update, can avoid locks
- Read-copy-update
  - Read structure to update
  - Make a local (shadow) copy
  - Update copy (without competition)
  - Change pointer from old copy to new copy
  - Release old copy when users are done
- Works well when changing link from old copy to new copy is atomic
- Useful for simple update type changes

COMMENTS ON SYNC MECHANISMS

Shared memory solutions – either software only or hardware supported (TSL or swap) have spin-locks; shared memory location can be accessed directly (read/written); “blocked” process consumes CPU cycles

Semaphores – requires OS support – scheduling based; semaphores CANNOT be accessed directly, only through UP() and DOWN(), and semaphore values are persistent; heavier cost than spin-lock code (system call); blocked process does not consume CPU cycles

Condition Variables – within monitor construct (language support needed); scheduling based (blocked process does not use CPU); condition “variables” are more like signals and queues – they are NOT persistent and can not be accessed directly, only through wait and signal

CLASSIC SYNCHRONIZATION PROBLEMS

Critical Region Problem: Mutual exclusion – only one process can be in Critical Region at a time ← next

Producer-Consumer Problem: Producers produce items and write them into buffers; Consumers remove items from buffers and consume them. A buffer can only be accessed by one process at a time.

Bounded Buffer Problem: P-C with finite # buffers

Readers-Writers: Any number of readers can read from a database simultaneously, but writers must access it solo

Dining Philosophers: Philosophers arranged in a circle share a fork between each pair of neighbors. Both forks are needed for a philosopher to eat, and only one philosopher can use a fork at a time.