Interprocess Communications

Interprocess Communications

- Motivations
- Shared Memory
- Mutual Exclusion and Semaphores
- Message Passing
- Classical IPC Problems

Motivations

- Processes (or threads) need to communicate to each other to complete a task, e.g., pipes, shared memory, sockets, …
  - They need interprocess communication (IPC).
- IPC is not only about communication, but more importantly about data synchronization.
  - When to read/write to ensure data consistency
- Also, all UNIX kernels are *reentrant* kernel.
  - User processes may be executing in kernel mode at the same time.
  - Reentrant kernels need data synchronization.

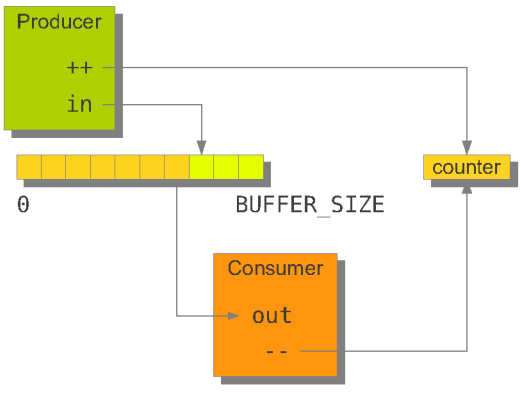
(cont'd.)

- Well-known IPC mechanisms are
  - Files
  - Sockets (byte streams)
    - Unix domain
    - Network
  - Pipes (of standard streams)
    - Anonymous or Named
  - Shared memory
  - Message queues, or mailboxes
  - Message passing (share nothing)
  - Semaphores

Shared Memory

- One of the most straightforward solution.
- Two or more processes share the same memory space so that they could communicate by reading/writing data from/to the memory space.
- Very efficient.
- Let's see a classic example: *the producer–consumer problem*
  - One process is a producer that write data to the shared buffer.
  - Another is consumer that read data from the shared buffer.

The Producer–Consumer Problem



(cont'd.)

```
Process P0 – Producer:

while (1) {
   /* buffer is full - wait */
   while (counter == BUFFER_SIZE);
   buffer[in] = produce ();
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```


(cont'd.)

```
Process P1 – Consumer:

while (1) {
   /* no data – wait */
   while (counter == 0);
   consume (buffer[out]);
   out = (out + 1) % BUFFER_SIZE;
   counter--;
}
```


So, we have solved the problem ?

- Both are correct separately.
- Process `P0` and `P1` share `counter` and `buffer`
- The counter controls when the process should access the buffer.
  - Its value is changed to reflect the number of items in the buffer.

counter++:	**counter--:**
`r0 = counter`	`r0 = counter`
`r0 = r0 + 1`	`r0 = r0 - 1`
`counter = r0`	`counter = r0`

- What's wrong with that ?

Race Conditions

- Let's consider the following sequence of execution when they run concurrently:

```
reg0 = counter     (reg0 = 5)
reg0 = reg0 + 1    (reg0 = 6)
// Interrupt occurred -- save reg0 to PCB
...
// Exit from ISR – schedule to run P1
reg0 = counter     (reg0 = 5)
reg0 = reg0 – 1    (reg0 = 4)
counter = reg0     (counter = 4)
// P0 continue – restore reg0 from PCB
counter = reg0     (counter = 6)
```

(cont'd.)

- The situation such that the final result depends on the order of executions is one of the most undesirable effect in execution and is called *race condition*.
- Race condition is **very** hard to debug.
- So, we learn that using shared memory may lead to race condition. How can we avoid that condition ?
- One of the solution is that to prohibit more than one processes to access the shared memory at the same time.
- In other words, we need *mutual exclusion*.

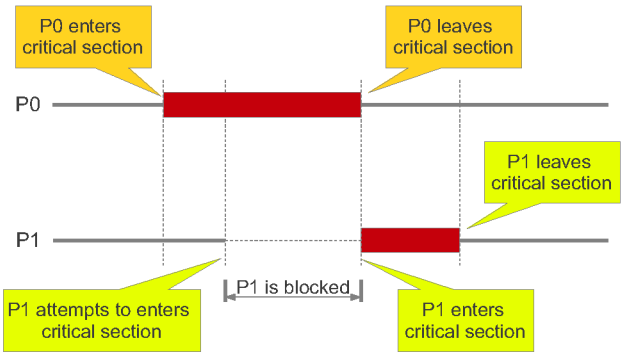
Mutual Exclusions

- Techniques to coordinate accessing the share memory among processes.
- One possible solution is that let each process to finish their execution on the shared memory one by one.
  - What if they have to wait results from one another ?
  - Performance ?
- A program consists of parts that access shared memory, files, doing critical things. These part are called *critical section* or *critical region*.
- The others are called *remainder section*.

(cont'd.)

- So, to avoid race condition, two or more processes must not execute code in their critical section at the same time.
  - This is good since the remainder section may be executed in parallel.
- A good solution should also hold the following conditions:
  - No assumption may be made about speed or number of CPUs.
  - No process running its remainder section may block other processes
  - No process should have wait forever to enter its critical section.

(cont'd.)



Disabling Interrupts

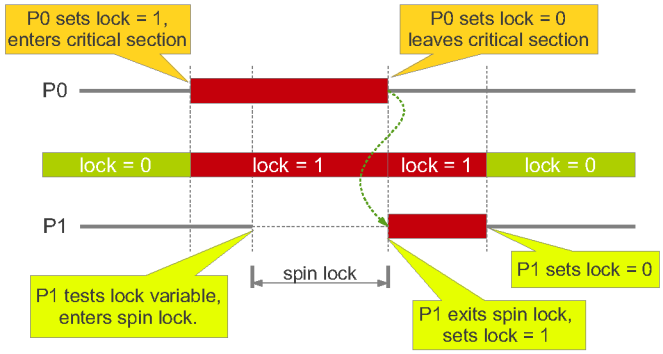
- One simple solution – disable interrupt when enter critical section, re-enable when exit.
  - So, no context switch – one process/thread would finish its job without any intervention.
- Well, it works, but ...
  - What if critical section is very long ?
    - Hardware freezing
  - On multiprocessor system, disabling interrupts on one processor won't work.
  - Disabling interrupts on all processors takes time.
  - Note that, disabling and re-enabling interrupts should be done carefully and only within kernel space by the operating system itself.

Lock Variables

- Use a shared variable to lock when enter critical section, so that the others will be blocked.

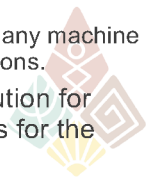
```
do {
  while (lock == 1);
  lock = 1;
  /* critical section */
  ...
  lock = 0;
  /* remainder section */
  ...
} while (1);
```


(cont'd.)



(cont'd.)

- There is a loop to wait condition on the lock variable to enter the critical section.
- This is called *busy waiting* and a lock that uses busy waiting is called a *spin lock*.
- The lock variable has to be shared among processes. So, the lock variable itself must be set or reset within critical section as well.
  - A single assignment statement requires many machine codes, and many more microcode executions.
- The lock variable is not an ultimate solution for mutual exclusion. Still, it is a good basis for the others.



Strictly Alternation

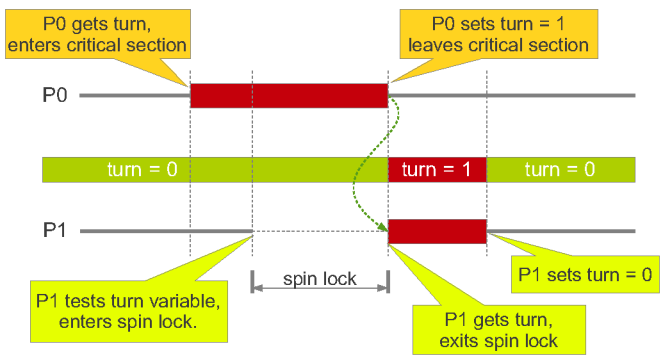
- Introduce a shared variable *turn* to indicate a process that may enter its critical section.

```
Process P0:
do {
  while (turn != 0);
  /* critical section */
  ...
  turn = 1;
  /* remainder section */
  ...
} while (1);
```


(cont'd.)

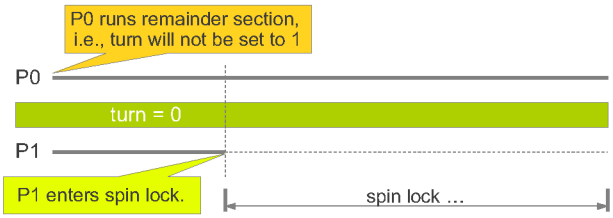
```
Process P1:
do {
  while (turn != 1);
  /* critical section */
  ...
  turn = 0;
  /* remainder section */
  ...
} while (1);
```


(cont'd.)



(cont'd.)

- It sets turn to leave critical section. So, only one process may execute its critical section.
- It strictly alternate between the two processes.
  - A process may be blocked by remainder section of another process.



Dekker's Algorithm

- Theodorus J. Dekker had combined the lock variable and the alternation approach.
  - Introduce a shared variable to indicate *attempt* to enter the critical section.

```
/* shared variables */
f0 = 0;
f1 = 0;
turn = 0;
```


(cont'd.)

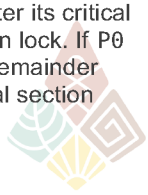
```
Process P0:
  f0 = 1;                 /* attempt to enter  */
  while (f1 == 1) {       /* check the others  */
    if (turn != 0) {      /* not my turn ?     */
      f0 = 0;             /* then, not enter   */
      while (turn != 0); /* and wait for turn */
      f0 = 1;             /* my turn, try again*/
    }
  }
  /* critical section */
  ...
  turn = 1;
  f0 = 0;
  /* remainder section */
  ...
```

(cont'd.)

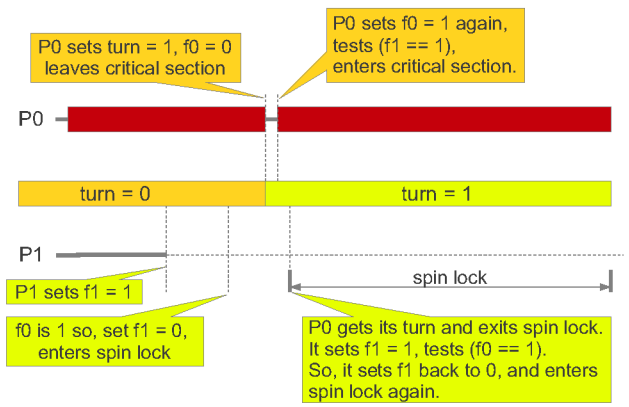
```
Process P1:
  f1 = 1;
  while (f0 == 1) {
    if (turn != 1) {
      f1 = 0;
      while (turn != 1);
      f1 = 1;
    }
  }
  /* critical section */
  ...
  turn = 0;
  f1 = false;
  /* remainder section */
  ...
```


(cont'd.)

- Dekker's algorithm is the *first* known corrected solution to the mutual exclusion problem.
  - Addressed by Edsger Wibe Dijkstra in 1965.
  - Can be extended to more than two processes.
- However, the algorithm cannot guarantee that a process will enter its critical section.
  - e.g., `P0` is in critical section, `P1` cannot enter its critical section so it sets `f1 = 0` and goes for spin lock. If `P0` exits the critical section and executes its remainder section quick enough, `P0` may enter critical section again. `P1` may have to wait indefinitely.



(cont'd.)



Peterson's Algorithm

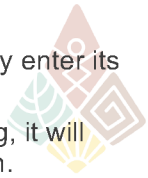
- Gary L. Peterson proposed a simpler solution in 1981:

```
int turn;
int flag[2] = {0, 0};

Process P0:
flag[0] = 1;
turn = 1;
while (flag[1] && turn == 1);
/* critical section */
...
flag[0] = 0;
/* remainder section */
...
```


(cont'd.)

```
Process P1:
flag[1] = 1;
turn = 0;
while (flag[0] && turn == 0);
/* critical section */
...
flag[1] = 0;
/* remainder section */
...
```



- If a process sets the flag, it will definitely enter its critical section.
- If there is only one process sets the flag, it will enter its critical section again and again.

Test-and-Set Lock Instruction

- Literally, the lock variable approach fails because it cannot test and set lock variable *atomically*.
- So, what if we do that in the hardware using only one instruction ?
  - Well, that will be atomic and that instruction is TSL.
- TSL works as the following:

```
tsl reg0, lock
```

  - Store `lock` to `reg0`.
  - Store non-zero to `lock`.
  - The operation is guaranteed to be indivisible.

(cont'd.)

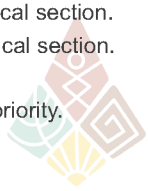
- TSL can be used to implement spin lock.

```
enter_critical_section:
  tsl reg0, lock  // reg0 ← lock, lock ← 1
  cmp reg0, 00h   // old lock == 0 ?
  jne enter_critical_section
  ret

leave_critical_section:
  mov lock, 00h   // unlock
  ret
```


Sleep and Wakeup

- So, we got the solutions
  - Software-based: Peterson's algorithm
  - Hardware-based: TSL instruction
- But, both are using busy waiting.
  - Busy waiting is very short. Still, it wastes CPU time.
  - It may introduce undesirable effects, e.g.,
    - P0 has a low priority and is running its critical section.
    - P1 has a high priority but cannot enter critical section.
    - P1 preempts P0 and runs spin lock.
    - P0 cannot continue because it has lower priority.
    - Both cannot go any further.
    - This is called *priority inversion problem*.



(cont'd.)

- To avoid busy waiting, one simplest solution is to use `sleep` and `wakeup` system calls.
  - Use `sleep` to block, instead of busy waiting.
  - Use `wakeup` to continue after exit the waiting.

Semaphores

- In 1965, E. W. Dijkstra suggested using an integer variable, called *semaphore*, to count the number of wakeups saved for future use.
- He also proposed two atomic operations on the semaphore:

```
void P(Semaphore S){
  while (S <= 0);
  S--;
}

void V(Semaphore S) {
  S++;
}
```


(cont'd.)

- Literally, semaphore represents the amount of resource left (or available).
- P stands for the made-up portmanteau word *prolaag*, short for *probeer te verlagen* or "try to decrease."
  - He is a dutch mathematician.
  - It tests condition and waits if the test fails.
  - Otherwise, it consume the resource, decrease the semaphore.
- V stands for *verhogen*, or "increase."
  - It releases the resource, increase the semaphore.

(cont'd.)

- Semaphore can be use to provide mutual exclusion, e.g.,

```
Semaphore S;
do {
  P(S);
  /* critical section */
  ...
  V(S);
  /* remaining section */
  ...
} while (1);
```

Mutex

- Since semaphore is used to count resources, it is call, well, *counting semaphore*.
- Most of the case, semaphore is used to lock and unlock resource.
- If the ability of counting is not required, the semaphore can be reduced to a simple one just to lock and unlock. This is called *binary semaphore*,
- Binary semaphore is essentially a variable for mutual exclusion, called *mutex*.

(cont'd.)

- Like semaphore, there are two atomic operations on the mutex:

```
void mutex_lock(Mutex M) {
  while (M == 0);
  M = 1;
}

void mutex_unlock(Mutex M) {
  M = 0;
}
```


(cont'd.)

- Again, like semaphore, the usage of mutex is, e.g.,

```
Mutex M;
do {
  mutex_lock(M);
  /* critical section */
  ...
  mutex_unlock(M);
  /* remaining section */
  ...
} while (1);
```

Monitor

- Still, semaphores have some undesirable effects like deadlock and starvation if they are used carelessly.
- In 1972, Sir Charles Antony Richard Hoare and Per Brinch Hansen proposed a new approach called *monitor*.
  - Sir C.A.R. Hoare invented the quicksort.
- A monitor is a special programming structure containing functions, variables, and internal data structures.
  - Processes may call functions inside a monitor but cannot directly access internal data.

(cont'd.)

- An example of monitor:

```
monitor foobar {
  integer i;
  boolean condition;

  function producer() {
    ...
  }

  function consumer() {
    ...
  }
}
```


(cont'd.)

- One important property of monitor is that it allows only one process to run inside the monitor.
  - It checks whether there exists a process inside the monitor.
  - If there is no process, the calling process will enter the monitor. Otherwise, it is blocked.
- So, there is only one process that may call either `producer()` or `consumer()`. This is essentially mutual exclusion.

(cont'd.)

- Monitor may use binary semaphore or mutex to check whether there exists a process inside the monitor.
  - Yes, there may be side effects. But, monitor is done at the level of programming language. So, monitor code is specially arranged by a compiler. It is less likely to have such side effects.
- Still, processes may be blocked, e.g.,
  - `P0` calls `producer()`, it enters the monitor and realize that the buffer is full. So, it has to wait.
  - `P1` calls `consumer()`, but cannot enter the monitor.
  - `P0` and `P1` cannot go any further.

(cont'd.)

- So, monitor introduces *conditional variables*. Let's use conditional variable to fix the previous problem:
  - `P0` calls `producer()`, it enters the monitor and realize that the buffer is full. So, it set a conditional variable, indicating that it has been blocked.
  - `P1` calls `consumer()`, the monitor checks conditional variable and found that a process (`P0`) has been blocked. So, `P1` may run safely within the monitor.
  - C. A. R. Hoare proposed to let `P0` be blocked, and allow `P1` to run within the monitor.
  - Brinch Hansen proposed to force `P0` to exit from the monitor before letting `P1` to run within the monitor.

(cont'd.)

- The main advantage of the monitors is that they make parallel programming much safer than semaphores.
- Still, a programming language must support monitor. This is the main disadvantage of the monitor since there is a small number of languages that support monitor.
  - e.g., Ada, C#, D, Delphi, Java, Modula-3, Python, Ruby, Squeak (smalltalk), µC++, ...

Message Passing

- An approach to communicate among processes without using shared memory
- Implemented in two system calls
  - `send` – to send a message
  - `receive` – to receive a message
- Widely used in parallel computing
  - e.g. Message Passing Interface (MPI)
- Two possible approaches
  - Direct – refer to process(es) directly
  - Indirect – refer to mailbox(es) or port(s) that link to process(es)

(cont'd.)

- Message passing can either be
  - Blocking – *synchronous*
    - Blocking send – the sender is blocked until the message is received.
    - Blocking receive – the receiver is blocked until the message is available.
  - Non-blocking (message queue) – *asynchronous*
    - Non-blocking send – the sender sends messages continuously.
    - Non-blocking receive – the receiver receives messages or null.

(cont'd.)

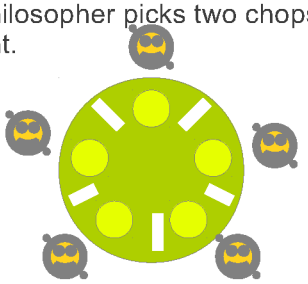
- Required mechanisms:
  - Acknowledgment
  - Authentication
  - Management
    - Network, memory, mailbox, ...

Classical IPC Problems

- Dining Philosophers Problem
- Reader and Writer Problem
- Sleeping Barber Problem

The Dining -Philosopher Problem

- n philosophers can either think or eat.
- There are n bowls and n chopsticks.
- To think, a philosopher touches nothing.
- To eat, a philosopher picks two chopsticks on his left and right.



(cont'd.)

```
do {
  P (chopstick[i]);
  P (chopstick[(i + 1) % n]);
  /* eat /
  ...
  V (chopstick[i]);
  V (chopstick[(i + 1) % n]);
  /* think */
  ...
} while (1);
```

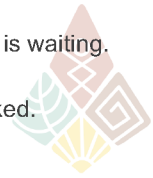
- This works, but might create a deadlock.
  - So, no philosopher can eat, they all starve – starvation!

(cont'd.)

- Any better solution to synchronize among them ?
  - Allow at most $n - 1$ philosopher to be sitting simultaneously at the table.
  - Pick/put both chopsticks in a critical section.
  - Use an asymmetric solution
    - an odd philosopher picks left chopstick, then right.
    - an even philosopher picks right chopstick, then left.
  - Dijkstra's solution
    - One philosopher picks right chopstick, then left.
    - The others picks left chopstick, then right.

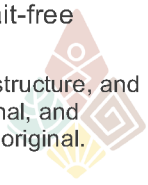
The Readers and Writers Problem

- Access shared objects, e.g., database
- All are readers, no problem, they can read simultaneously.
- One writer with the others, big problem, inconsistencies.
  - Writer must have exclusive access to the objects.
- Two variations:
  - All readers are not blocked, even a writer is waiting.
    - The writer may starve.
  - If a writer is waiting, the readers are blocked.
    - The reader may starve.

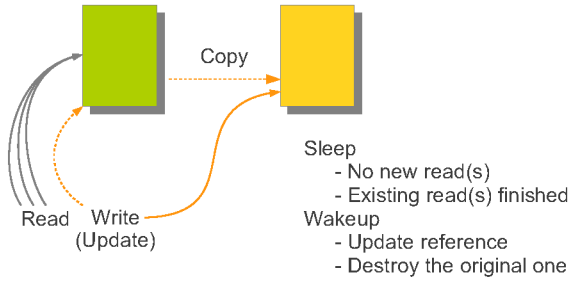


(cont'd.)

- A possible solution to avoid starvation:
  - The writer is blocked until there is no reader using the objects.
  - If there exists the writer, the others – both readers and writers, must be blocked until the writer completes its object manipulations.
  - The writer manipulates objects in a critical section.
- Read-Copy-Update (RCU) can be a wait-free reader solution.
  - It can be expensive to copy the updating structure, and maintain two copies of the structure (original, and updated) until there is no reference to the original.



(cont'd.)



- RCU is covered by U.S. patent, assigned to Sequent Computing Systems in 1995.

The Sleeping Barber Problem

- One barber, one barber chair, n waiting chairs.
- If no customer, the barber sleeps on the barber chair.
- If there is one customer, the barber must wake up and cut the customer's hair.
- If additional customers enter, they either sit down or leave if there is no chair available.
- How do we prevent the race condition ?
  - Three semaphores:
    - `mutex` to perform mutual exclusion
    - `barber` to get/release barber chair
    - `customer` to get/release waiting chair

(cont'd.)

```
/* number of waiting chairs */
#define CHAIRS 5

/* counter for waiting customers */
semaphore customer = 0;

/* is barber chair occupied ? */
semaphore barber = 0;

/* mutual exclusion */
semaphore mutex = 1;

/* required to count semaphore */
int waiting = 0;
```

(cont'd.)

```
void barber (void) {
  while (1) {
    P(customer);  /* wait for customer      */
    P(mutex);     /* enter critical section */
    wait--;       /* one less waiting       */
    V(barber);    /* acquire barber chair   */
    V(mutex);     /* exit critical section  */
    cut_hair ();  /* cut hair – remainder   */
  }
}
```


(cont'd.)

```
void customer (void) {
  P(mutex);        /* enter critical section */

  if (waiting < CHAIRS) {
    /* available chair – wait */
    waiting++;     /* sit and wait */
    V(customers);  /* one more customer */
    V(mutex);      /* exit critical section */
    P(barber);     /* wait to get haircut */
    get_haircut(); /* get hair cut */
  } else {
    /* no available chairs – exit */
    V(mutex);       /* exit critical section */
  }
}
```

IPC in Linux

- Anonymous Pipes and Named pipe (FIFO)
  - See `man pipe`
- Semaphores
  - See `kernel/semaphore.c`
- Mutexes
  - See `kernel/mutex.c`
  - See `Documentation/mutex-design.txt`
- Shared memory (shm)
- Fast user-space mutexes (Futexes)
- Sockets
- POSIX Messages Queues