

-
-
-
-
-
-
-
-
-
-

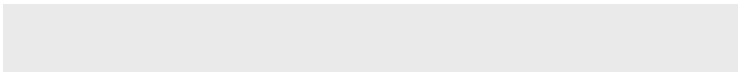


THE UNIVERSITY OF
WESTERN AUSTRALIA

Computer Operating Systems 214



6. CONCURRENCY



-
-
-
-
-
-
-
-

Copyrights on Figures

- Figures presented in these notes with the following captions are extracted from the textbook and copyrighted by the publisher as indicated
 - (OS3e) W. Stallings, “*Operating Systems: Internals and Design Principles*”, 3rd Edition, Prentice-Hall, 1997
 - (osdi2) A.S. Tanenbaum & A.S Woodhull, “*Operating Systems: Design and Implementation*”, 2nd Edition, Prentice-Hall, 1997
- **DISCLAIMER**
 - Whilst every attempt is made to maintain the accuracy and correctness of these notes, the author, Dr. Roberto Togneri, makes no warranty or guarantee or promise express or implied concerning the content.

Issues of Concurrency

- Many processes running concurrently (in parallel)
 - Uni-processor
 - M processes with pre-emptive scheduling
→ M processes run effectively concurrently (pseudo-parallelism)
 - Multi-processor
 - N processors → N processes running concurrently
- Characteristics of concurrent processes
 - Processes execute code at different speeds
 - CPU scheduling, I/O events
 - Per process instruction execution predictable (A1, A2, A3, A4), BUT many process instruction execution unpredictable {(A1,B1,A2,B2,A3,B3),(A1,A2,A3,B1,B2,B3)}
 - Processes execute independently of one another
- Problems with concurrent processes
 - Processes may need to access a **shared resource**
 - Who does what and when and what is the final result?
 - Processes may need to cooperate and **synchronise**
 - When process A formats file process B then has to print it
→ A needs to tell B it has a file ready to print and B has to wait for the file

Issues of Concurrency

- Processes accessing a shared resource

- *Example:* Process A and B both execute:
LOAD SV
ADD 1
STORE SV
and share the variable SV

- Scenario 1 (assume SV initially 5)

- A: LOAD SV
 - A: ADD 1
 - A: STORE SV
 - B: LOAD SV
 - B: ADD 1
 - B: STORE SV → SV is now 7

- Scenario 2 (assume SV initially 5)

- A: LOAD SV
 - B: LOAD SV
 - A: ADD 1
 - A: STORE SV
 - B: ADD 1
 - B: STORE SV → SV is now 6

- **Race Condition**

- Final result depends on the order of execution (*who wins the race?*)

Mutual Exclusion

- Solution to shared access problem
 - **Critical section**
 - Part of program code that uses a shared resource
 - **Mutual Exclusion Principle**
 - Control processes when entering and leaving the critical section such that only one process is allowed to be in the critical section at any one time
 - Critical Section (*Mutual Exclusion*) Problem
 - How should processes entering the critical section be controlled so as to avoid a race condition?
 - The critical section of a program can be executed safely (without causing a race condition) by multiple processes or threads → **re-entrant code**
 - Critical Section (*Mutual Exclusion*) Solution Requirements
 - Mutual Exclusion: Must satisfy mutual exclusion principle
 - Progress: If a process wants to enter the critical section and no other processes want to then it should enter unimpeded. If all processes wanting to enter the critical section are prevented from doing so forever → **deadlock**
 - Bounded Waiting: If many processes want to enter the critical section then there must be a bound as to how long any process needs to wait in order to enter the critical section. If a process waits indefinitely → **starvation**

Software Solution: Take 1

- 2 process solution (process 0 and process 1)
- Both processes need to execute the following code in the most general case:

- Process 0 has local variables *me*=0 and *other*=1
Process 1 had local variables *me*=1 and *other*=0
- *int turn = 0; /* shared variable, initially set to 0 */*

```
....  
while (TRUE) {  
    while (turn != me)  
        ; /* wait */  
    critical_region(); /* go for it! */  
    turn = other;  
    noncritical_region();  
}
```

- Does solution meet requirements
 - Mutual Exclusion: *Yes!* The *turn* variable can only have one value, so only one process can enter the critical section
 - Progress: *No!* **strict alternation** prevents process 0, say, from entering critical_section again without process 1 having done so first (e.g. process 0 can't enter consecutively N times before process 1 enters again)
 - Bounded Waiting: *No!* strict alternation makes process 0 dependent on process 1 and process 1 may force process 0 to starve

Software Solution: Take 2

- 2 process solution (process 0 and process 1)
- Both processes need to execute the following code in the most general case:

- Process 0 has local variables $me=0$ and $other=1$
Process 1 had local variables $me=1$ and $other=0$
- `int interested[2];` */* shared variable */*

```
...
while (TRUE) {
    interested[me] = TRUE;
    while (interested[other] == TRUE)
        ; /* wait */
    critical_region();
    interested[me] = FALSE;
    noncritical_region();
}
...
```

- Does solution meet requirements?
 - Mutual Exclusion: *Yes!* Setting `interested[me] = TRUE;` prevents other process entering the critical section.
 - Progress: *No!* If both processes executed `interested[me] = TRUE;` concurrently then they will both loop indefinitely in the while (...) loop → **deadlock**. But there is *no strict alternation* (e.g. process 0 can enter N times before process 1 enters again)
 - Bounded Waiting: *No!* Deadlock technically means **starvation**

Peterson's (*Software*) Solution

- 2 process solution (process 0 and process 1)
 - Combines the best of the Take 1 and Take 2 solutions
- Both processes need to execute the following code in the most general case:

```
– Process 0 has local variables me=0 and other=1
  Process 1 had local variables me=1 and other=0
– int turn, interested[2];      /* shared variables */
...
  while (TRUE) {
    interested[me] = TRUE;
    turn = other;
    while (turn == other && interested[other] == TRUE)
      ; /* wait */
    critical_region();
    interested[me] = FALSE;
    noncritical_region();
  }
...
```

- Does solution meet requirements?
 - Mutual Exclusion: *Yes!* Combination of Take 1 and Take 2
 - Progress: *Yes!* **No strict alternation** because of *interested*[*me*] = *FALSE*; allows other process to always enter. **No deadlock** because of shared *turn* variable.
 - Bounded Waiting: *Yes!* Once process sets *interested*[*me*] = *TRUE*; then it will be guaranteed to enter after no more than one entry by the other process.

Hardware Solution: Disable Interrupts

- Disable Interrupts

- Sample code

- ...
disable_interrupts();
critical_region();
enable_interrupts();
...

- Features

- No clock interrupts occur and process has exclusive access to the CPU while in the critical section → no other process can interfere
 - Simple and efficient → used by the kernel for **atomic** [i.e. operation must run to completion without pre-emption] or indivisible execution of code

- Problems

- Disabling pre-emptive scheduling adversely affects system performance (e.g. *critical_region()* may take minutes to execute)
 - Users should not be able to disable and enable CPU interrupts at will
 - e.g. user may easily forget to enable interrupts and system “locks up”
 - Doesn't work for multi-processors since only one CPU is affected

Hardware Solution: TSL instruction (*)

- Test and Set Lock (TSL) instruction
 - **tsl A, lock**
 - Copies contents of lock variable (stored in memory) to CPU register A ($A = lock$)
 - Sets lock variable to a non-zero value ($lock = 1$)
 - Operation is guaranteed atomic for N processors accessing a common memory
 - When process executes the TSL instruction it must run to completion or be restarted if an interrupt occurs
 - For N processors the bus is locked for the duration of the TSL instruction, thus concurrent access to the lock variable memory is impossible
 - Using the TSL instruction to solve the mutual exclusion problem
 - *TSL_enter_region:*

```
tsl A, lock
cmp A, #0
jne TSL_enter_region
ret
```
 - *TSL_leave_region:*

```
move lock, #0
ret
```
 - ...

```
move lock, #0 ; initialise lock to #0
call TSL_enter_region
call critical_section
call TSL_leave_region
...
```
 - Bounded waiting may not be satisfied
 - Processes compete to execute TSL and low priority processes may be starved

Busy Wait and Producer-Consumer

- The **busy wait** problem
 - Peterson's Solution and TSL instruction involve busy waiting (polling)
 - Process keeps looping (*while(...)* or *jne enter_region*) waiting to enter critical section → CPU is used to do this → what a waste!
 - High priority process may have to wait indefinitely for a lower priority process to leave critical section since it is pre-empting the CPU from that process for polling!
 - Solution? Processes should block not busy wait
 - Needs to be implemented by the OS → **semaphores** and **monitors**
- Classic Problem: Producer-Consumer
 - Shared bounded buffer of data
 - *Producer process*: acquires or generates data and places it in buffer
 - *Consumer process*: consumes or uses data from buffer
 - Implementation by circular buffer of size N
 - Producer pointer (*in*): indicates next empty slot in buffer to use, incremented as slots are filled ($in = (in + 1) \% N$)
 - Consumer pointer (*out*): indicates next full slot in buffer to use, incremented as slots are consumed ($out = (out + 1) \% N$)
 - Observation: " $in > out$ ", since consumer can't overtake producer!

Producer-Consumer

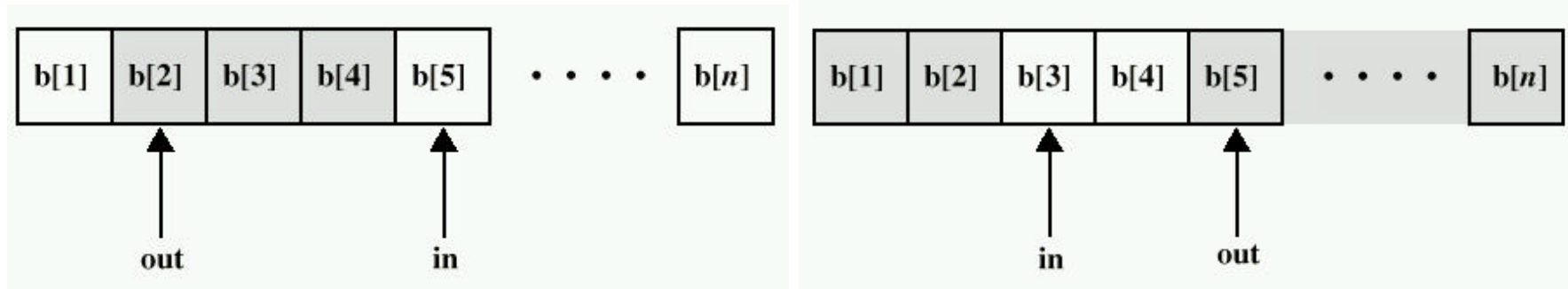


Figure 5.16, OS3e: *Producer-Consumer buffer*

- Producer-Consumer applications
 - I/O and network buffering
- Concurrency control problem
 - Mutual exclusive access of buffer by producer and consumer
 - Only one at a time can access buffer, strictly not necessary if $out \neq in$, but simpler solution
 - Producer must block if buffer is full ($|in - out| = N$)
 - Consumer must block if buffer is empty ($out = in$)
- Concurrency control solution
 - Simple mutual exclusion solutions will not work as producer/consumer will block when in the critical section
 - If producer blocks, only the consumer can wake it when it consumes an item and buffer is no longer full, but consumer needs to enter critical section to do this!
→ Classic **deadlock** scenario

Signalling and Semaphores

- Inter-process communication by signal / wait
 - Mutual exclusion and process synchronisation without busy wait
 - Process A executes *wait()*; and blocks until another process issues a *signal()*;
 - Processes can synchronise (A waits for B to signal it when it has arrived)
 - Mutual exclusion can be achieved (A waits to enter, B signals A when it finishes)
- Semaphore
 - Proposed by Dijkstra (1965)
 - Performs signalling using a special shared semaphore variable (s)
 - $s = \{0,1,2,\dots,n\} \rightarrow$ **counting semaphore**
 - $s = \{0,1\} \rightarrow$ **binary semaphore**
 - Semaphore operations
 - **down(&s)** (also wait(s))
 - *if (s == 0) then block else s = s - 1;*
 - **up(&s)** (also signal(s))
 - *if (processes are blocked on down(s)) then wakeup one of the blocked processes*
 - else*
 $s = s + 1$
 - down(&s) and up(&s) operations are atomic

Semaphores: Implementation

- Implementation of semaphores

- $s \rightarrow$ struct semaphore {int count; int value; QTYPE queue}
- down(&s)
{
 disable_interrupts() OR TSL_enter_region
 s.count = s.count - 1;
 if (s.count < 0) {
 Place this process in s.queue
 enable_interrupts() OR TSL_leave_region
 Block (wait for wakeup)
 }
 else enable_interrupts() OR TSL_leave_region
}
- up(&s)
{
 disable_interrupts() OR TSL_enter_region
 s.count = s.count + 1;
 if (s.count <= 0) {
 Select a process P from s.queue
 Wakeup process P
 Place process P on ready queue
 }
 enable_interrupts() OR TSL_leave_region
}
- s.count can take on negative value
 - semaphores are usually defined as non-negative (s.value = 0, if s.count < 0)
- Mutual exclusion when performing semaphore operations
 - TSL instruction
 - suitable for multi-processors
 - busy wait
 - disable interrupts
 - not suitable for multi-processors
 - does not involve a busy wait
- Processes are removed from s.queue in FIFO order

Semaphores: Mutual Exclusion

- Mutual exclusion with semaphores
 - Main process declares and initialises a binary semaphore:
 - `semaphore s = 1; /* binary semaphore */`
 - All processes then execute:
 - ...
 - ...
 - `down(&s);`
 - `critical_region();`
 - `up(&s)`
 - ... `/* that's it! */`
 - *Bonus!* solution for N processes and no busy wait
 - Does solution satisfy requirements?
 - Mutual Exclusion: *Yes!* First process that does `down(&s)` is allowed through, other processes then see $s = 0$ and block
 - Progress: *Yes!* Any one process can enter as many times as it likes with no waiting, and at least one process will be in the critical region or be allowed in → *no deadlock*
 - Bounded Waiting: *Yes!* Blocked processes are woken up in FIFO order → *no starvation*

Semaphores: Synchronisation

- Process A waits for B

- Main process declares and initialises a binary semaphore

- *semaphore event = 0;*

- Process A

- ...
down(&event); / sync point */*
...

Process B

...
up(&event); / sync point */*
...

- Process A and B wait for each other

- Main process declares and initialises binary semaphores

- *semaphore ev1 = ev2 = 0;*

- Process A

- ...
up(&ev1);
down(&ev2); / sync point */*
...

Process B

...
up(&ev2);
down(&ev1); / sync point */*
...

- More complex solutions for more than two processes

Semaphores: Producer-Consumer (*)

```
#define N 100          /* number of slots in the buffer */

semaphore mutex = 1; /* controls access to critical region */
semaphore empty = N; /* counts empty buffer slots */
semaphore full = 0;  /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) { /* TRUE is the constant 1 */
        produce_item(&item); /* generate something to put in buffer */
        down(&empty); /* decrement empty count */
        down(&mutex); /* enter critical region */
        enter_item(item); /* put new item in buffer */
        up(&mutex); /* leave critical region */
        up(&full); /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) { /* infinite loop */
        down(&full); /* decrement full count */
        down(&mutex); /* enter critical region */
        remove_item(&item); /* take item from buffer */
        up(&mutex); /* leave critical region */
        up(&empty); /* increment count of empty slots */
        consume_item(item); /* do something with the item */
    }
}
```

- Figure 2.12, osdi2: *Producer-Consumer using semaphores*

Monitors

- Problem with semaphores
 - Too low-level → too easy to make a mistake which is too difficult to detect
 - change order of down(&s) and up(&s) or forget one or the other
 - deadlock → processes hang for no apparent reason!
 - race condition → the answer is always different!
- Monitor
 - Protected object
 - Has its own local data and procedures which cannot be accessed directly
 - Mutual Exclusion operation
 - Monitor object can only be in execution by one process at any one time
 - Processes attempting to enter monitor will block until monitor is available
 - Synchronisation primitives
 - Process can do a *wait(cond)* on the special monitor condition variable
 - process is blocked and the monitor is released
 - Process can do a *signal(cond)*
 - process is then suspended or exits monitor and one of the *wait(cond)* processes is given the monitor so that it resumes execution → if no process is waiting then no action is taken
 - Implementation controversy
 - Process must exit after doing a *signal(cond)* → Hansen (1975)
 - Process is suspended after doing a *signal(cond)* → Hoare (1974)
 - Programming languages (not OS!) with monitor construct
 - Concurrent Pascal, Pascal-plus, Modula-2, Modula-3

Monitors: Implementation

- *Blocked semaphore queues*
 - Queues: entering monitor, one per condition variable, urgent signal
 - Compiler uses combination of semaphore constructs (which are supported by OS) to manipulate processes leaving/entering monitor and the various queues
 - Compiler use of semaphores is correct (unless compiler is still in beta test!)

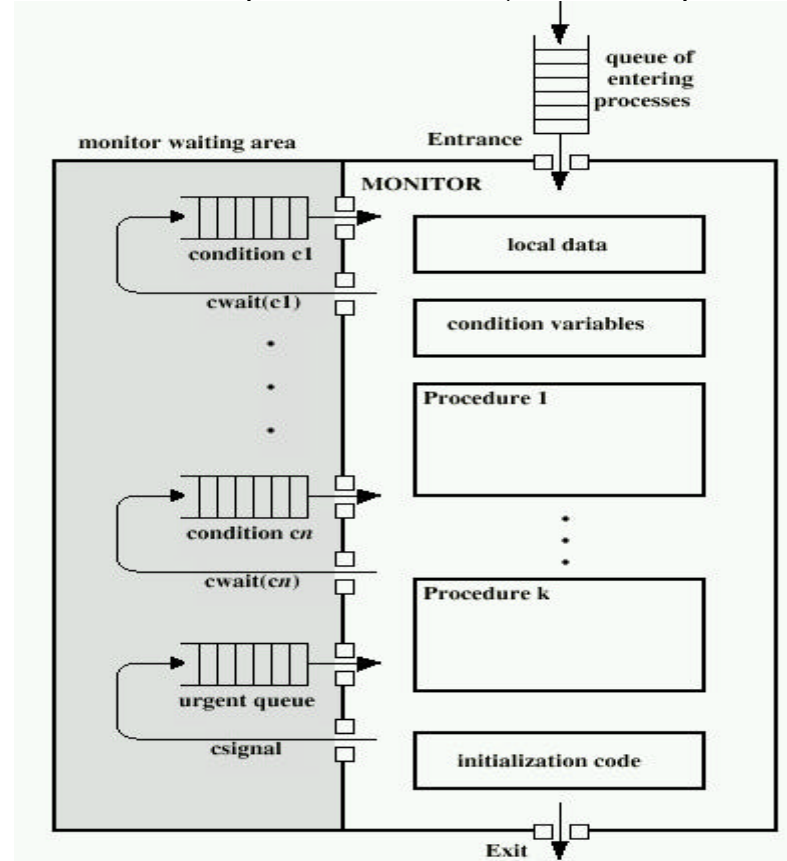


Figure 5.22, OS3e: Structure of a monitor

Monitors: Producer-Consumer

(* monitor declaration and definition *)

monitor ProducerConsumer

condition full, empty;

integer count;

procedure enter(item);

begin

if count = N **then wait**(full);

enter_item(item);

count := count + 1;

if count = 1 **then signal**(empty)

end;

procedure remove(item);

begin

if count = 0 **then wait**(empty);

remove_item(item);

count := count - 1;

if count = N-1 **then signal**(full)

end;

count := 0; {executed with the first invocation}

end monitor;

(* procedures using monitor *)

procedure producer;

begin

while true **do**

begin

produce_item(item);

ProducerConsumer.enter(item)

end

end;

procedure consumer;

begin

while true **do**

ProducerConsumer.remove(item);

consume_item(item)

end

end;

(* main program *)

begin

parbegin

producer; consumer

parend

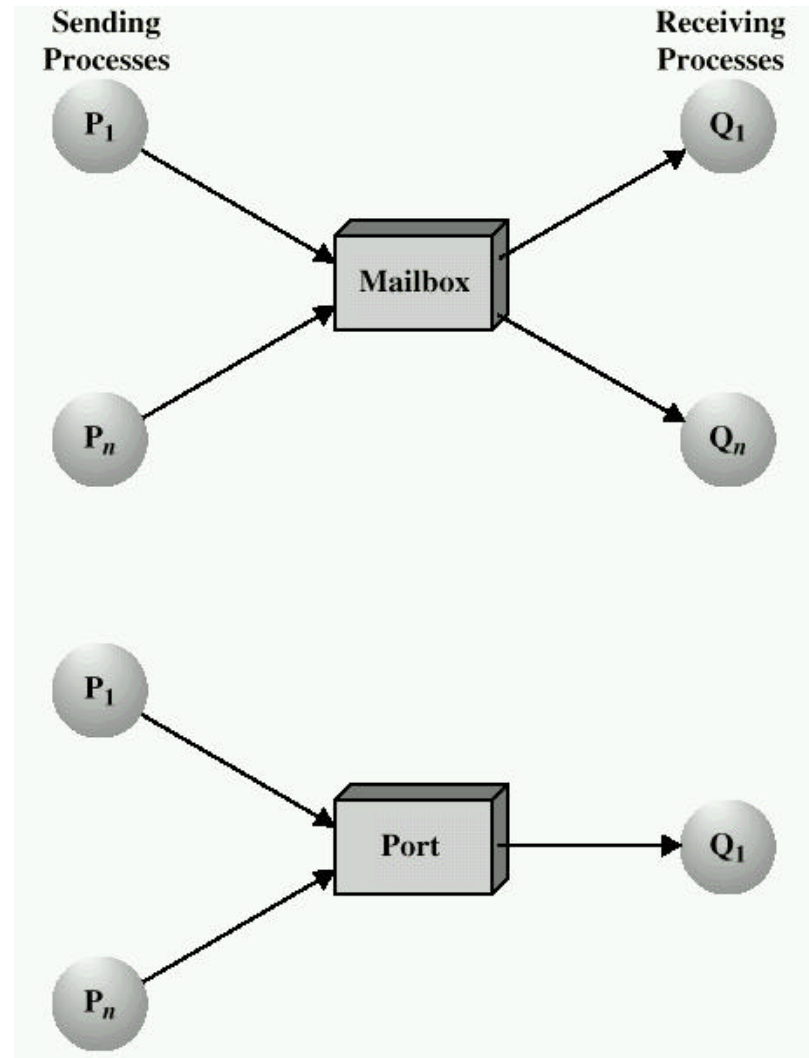
end

- Figure 2.14, osdi2: *Producer-Consumer with monitor*

Message Passing

- Problem with semaphores and monitors
 - Relies on underlying OS to ensure mutual exclusion and protection
 - OK for Uni-processor systems
 - OK for Multi-processor systems with shared memory (i.e. Symmetric Multi-Processors (SMP))
 - Cannot be easily extended to concurrency problems in distributed systems
 - Workstation cluster computing → Network of Workstations (NoW)
 - Multi-processors with distributed memory
 - Distributed client-server interactions
- Message Passing
 - Non-blocking *send()*, but blocking *receive()* primitives
 - *send(dest, msg)* → messages sent are queued and there is no need to wait
 - *receive(dest, msg)* → if queue is empty process blocks waiting for a message
 - Restricted to resolving concurrency problems in distributed systems
 - Message passing processing → too many overheads and not very efficient
 - Unreliable network → lost messages can create deadlock situations

Message Passing Implementations



• Figure 5.24, OS3e: *Indirect Process Communication*

Message Passing: Mutual Exclusion (*)

- Use mailbox form of message passing

- Main process creates and initialises mailbox

- `create_mailbox(mutex);` */* mutex is the mailbox destination */*
`send(mutex, msg);`

- All processes execute:

- ...
`receive(mutex, msg);` */* msg can contain dummy data */*
`critical_region();`
`send(mutex, msg);`
...

- Use port form of message passing

- *Lock server* process listens on port LOCK

- Simulates mutual exclusion operation for distributed processes
 - Relies on messages containing the source ID so server can identify client requests

- All processes execute:

- ...
`send(LOCK, "down");` */* indicate to lock server the semaphore down operation */*
`receive(LOCK, "go");` */* wait for lock server to allow access */*
`critical_region();`
`send(LOCK, "up");` */* indicate to lock server the semaphore up operation */*
...

Message Passing: Producer-Consumer

```
#define N 100                /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                /* message buffer */

    while (TRUE) {
        produce_item(&item);    /* generate something to put in buffer */
        receive(consumer,&m);    /* wait for an empty to arrive */
        build_message(&m,item); /* construct a message to send with packed item*/
        send(consumer,&m);      /* send item to consumer */
    }
}

void consumer(void)
{
    int item,i;
    message m;

    for(i=0; i < N; i++)
        send(producer,&m);      /* send N empties */
    while (TRUE) {
        receive(producer,&m);    /* get message containing item */
        extract_item(&m,&item); /* extract item from message */
        send(producer,&m);      /* send back empty reply */
        consume_item(item);     /* do something with the item */
    }
}
```

- Figure 2.15, osdi2: *Producer-Consumer with message passing*

Classic Problem: Readers-Writers

- Multiple readers, Single writers
 - Multiple reader processes are allowed to access shared resource
 - Single writer process is allowed to access shared resource, no other readers or writers are allowed → mutual exclusive access
 - Readers have priority
 - As long as there are readers, writer process will not be allowed access and can starve
 - Simplest solution but unfair
 - Writers have priority
 - A writer process will block subsequent reader processes and be allowed exclusive access after the last reader leaves
 - More complex solution but fairer
- Applications
 - Database Transaction Processing
 - Database is shared resource
 - Readers → database queries (e.g. bank account balance, Alta-Vista search)
 - Writers → database updates (e.g. bank transfer, Alta-Vista “spider” WWW robot)
- Multiple readers, Multiple writers
 - Allow multiple writers to different parts of shared resource

Readers-Writers: Semaphore Solution (*)

- Readers have priority

```
semaphore mutex = 1;      /* controls access to 'rc' */
semaphore db = 1;        /* controls access to the data base */
int rc = 0;              /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {        /* repeat forever */
        down(&mutex);     /* get exclusive access to 'rc' */
        rc = rc + 1;      /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader */
        up(&mutex);       /* release exclusive access to 'rc' */
        read_data_base(); /* access the data */
        down(&mutex);     /* get exclusive access to 'rc' */
        rc = rc - 1;      /* one reader fewer now */
        if (rc == 0) up(&db); /* if this is the last reader ... */
        up(&mutex);       /* release exclusive access to 'rc' */
        use_data_read();  /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {        /* repeat forever */
        think_up_data();  /* noncritical region */
        down(&db);        /* get exclusive access */
        write_data_base(); /* update the data */
        up(&db);          /* release exclusive access */
    }
}
```

- Figure 2.19, osdi2: Readers-Writers with semaphores

Readers-Writers: Monitor Solution (*)

```
monitor ReadersWriters
condition ReadingAllowed, WritingAllowed;
boolean SomeoneIsWriting;
integer reader;

procedure BeginReading
begin
  if SomeoneIsWriting or queue(WritingAllowed)
    then wait(ReadingAllowed);
  readers := readers + 1;
  signal(ReadingAllowed)
end;

procedure FinishedReading
begin
  readers := readers - 1;
  if readers = 0 then signal(WritingAllowed)
end;

procedure BeginWriting
begin
  if readers > 0 or SomeoneIsWriting
    then wait(WritingAllowed);
  SomeoneIsWriting := TRUE
end

procedure FinishedWriting
begin
  SomeoneIsWriting := FALSE;
  if queue(ReadingAllowed)
    then signal(ReadingAllowed)
    else signal(WritingAllowed)
  end;

  readers := 0;
  SomeoneIsWriting := FALSE
end monitor;
```

(* procedures using monitor *)

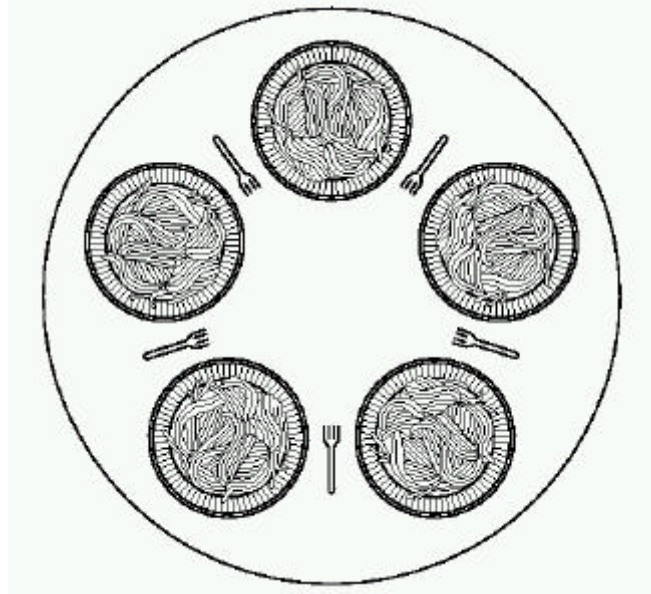
```
procedure writer;
begin
  while true do
    begin
      ReadersWriters.BeginWriting;
      Write_DataBase();
      ReadersWriters.FinishedWriting
    end
  end;

procedure reader;
begin
  while true do
    ReadersWriters.BeginReading;
    Read_DataBase();
    ReadersWriters.FinishedReading
  end
end;
```

Writers have priority

Classic Problem: Dining Philosophers

- 5 philosophers (*the* classic problem from Dijkstra in 1965)



• Figure 2.16, osdi2: *Dining Philosophers Table*

- Each philosopher is either thinking or eating
- Both a left and right fork are needed in order to eat
 - No more than 2 philosophers sitting on opposite sides of table can eat concurrently
- Applications
 - Modelling processes (philosophers) that are competing for access to a limited number of resources (forks)

Dining Philosophers: Non-Solutions

- Non-Solution 1

- When hungry each philosopher executes:

- take_fork(LEFT); */* block if fork is not available */*
take_fork(RIGHT);
eat();
put_fork(LEFT);
put_fork(RIGHT);

- *Problem?* Deadlock occurs if all philosophers pick up their LEFT fork together

- Non-solution 2

- When hungry each philosopher executes:

- down(&mutex);
take_fork(LEFT); */* forks available since only one philosopher in critical section */*
take_fork(RIGHT);
eat();
put_fork(LEFT);
put_fork(RIGHT);
up(&mutex);

- *Problem?* Inefficient since only one philosopher is allowed to eat at any one time

Dining Philosophers: Semaphore Solution

```
#define N 5 /* number of philosophers */
#define LEFT (i-1)%N /* number of i's left neighbor */
#define RIGHT (i+1)%N /* number of i's right neighbor */
#define THINKING 0 /* philosopher is thinking */
#define HUNGRY 1 /* philosopher is trying to get forks */
#define EATING 2 /* philosopher is eating */

int state[N]; /* array to keep track of everyone's state */
semaphore mutex = 1; /* mutual exclusion for critical regions */
semaphore s[N]; /* one semaphore per philosopher */

void philosopher(int i) /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) { /* repeat forever */
        think(); /* philosopher is thinking */
        take_forks(i); /* acquire two forks or block */
        eat(); /* yum-yum, spaghetti */
        put_forks(i); /* put both forks back on table */
    }
}
```

- Figure 2.18(a), osdi2: *Dining Philosophers using semaphores*

Dining Philosophers: Semaphore Solution (*)

```
void take_forks(int i)      /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);          /* enter critical region */
    state[i] = HUNGRY;     /* record fact that philosopher is hungry */
    test(i);              /* try to acquire 2 forks */
    up(&mutex);            /* exit critical region */
    down(&s[i]);           /* block if forks were not acquired */
}

void put_forks(int i)      /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);          /* enter critical region */
    state[i] = THINKING;  /* philosopher has finished eating */
    test(LEFT);           /* see if left neighbor can now eat */
    test(RIGHT);          /* see if right neighbor can now eat */
    up(&mutex);           /* exit critical region */
}

void test(int i)           /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY &&
        state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING; /* Yes! the forks are free for philosopher i */
        up(&s[i]);         /* signal philosopher i that he can now eat */
    }
}
```

- Figure 2.18(a), osdi2: Dining Philosophers using semaphores

Case Study: UNIX System V

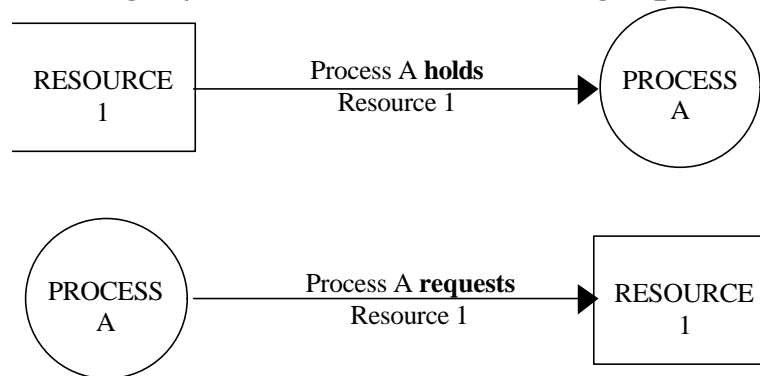
- UNIX System V
 - Pipes
 - FIFO circular buffer with producer-consumer interaction
 - producer blocks if pipe is full, consumer blocks if pipe is empty
 - Unnamed pipe
 - *pipe()* : only by the same process and children
 - Named pipe
 - *mkfifo()* : unrelated processes → creates special file of with file type 'p'
 - Similar to UNIX DOMAIN sockets
 - Message Queues
 - Block of text with accompanying type → acts like mailbox
 - *msgsnd()* : send message of specified type, block if queue is full
 - *msgrcv()* : retrieve message of specified type, block if queue is empty
 - Shared Memory
 - Processes can synchronise by access to shared memory variable
 - *shmget()* : allocate shared memory segment
 - *shmat()* : attach process to shared memory segment
 - *shmdt()* : detach process from shared memory segment
 - Requires use of semaphores for mutual exclusive access

Case Study: Windows NT

- Semaphores
 - Generalisation of semaphore definition → more flexible
 - *semctl()*: semaphore control operation
 - *semget()*: allocate a semaphore
 - *semop()*: semaphore operations
- Windows NT
 - Synchronisation Objects
 - Unsignalled state
 - thread will suspend when acquiring object handle in this state
 - Signalled state:
 - all or some suspended threads will be released when object enters signalled state
 - different events can cause object to enter the signalled state
 - Mutual exclusion uses Mutex or Mutant object
 - Object enters signalled state when the owning thread or any other thread releases the object, then one thread currently suspended on the object will be released
 - Other objects
 - Timer: acts as interval timer (countdown or **watchdog timer**)
 - Event: client-server synchronisation
 - Semaphore: regulates number of threads that can use a resource

Issues of Deadlock

- Computer systems are full of dedicated resources
 - Each resource can only be used by one process at a time
 - plotters, CD-ROM, DAT tape, printer, CPU, memory, scanner, etc.
 - Types of resources
 - *Preemptible*: process can release resource even though it hasn't finished
 - round-robin scheduling (CPU), process memory swapping (memory)
 - *Non-preemptible*: process cannot release resource until it has finished
 - printer (can't mix output to printer from different printer processes!)
 - Necessary (but not sufficient) conditions for deadlock
 - *Mutual Exclusion*: Only one process may use a resource at a time
 - *Hold-and-wait*: A process currently holding resources can request new resources
 - *No preemption*: Resources cannot be forcibly removed from a process until released by process
 - Deadlock modelling by resource allocation graph

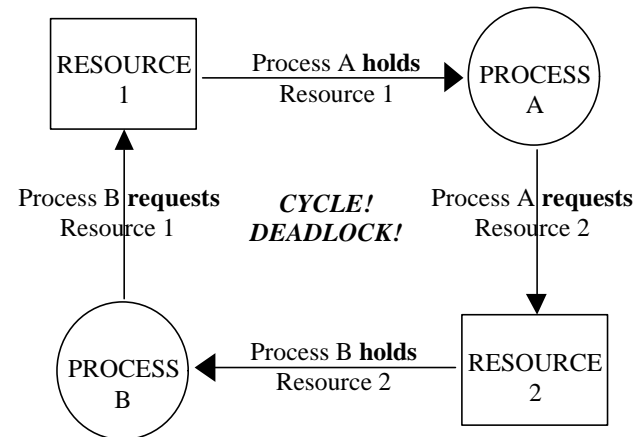
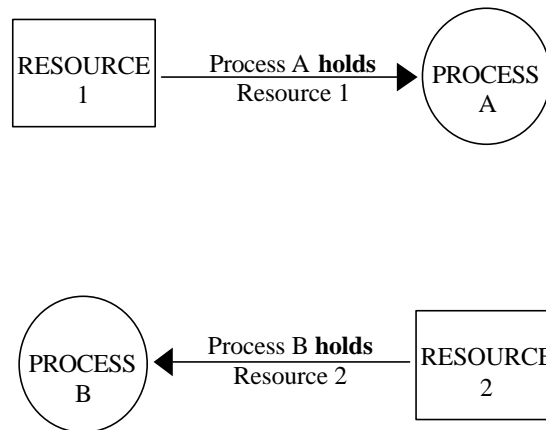


Condition for Deadlock

- Sufficient and Necessary conditions for deadlock
 - In addition to the 3 necessary (but not sufficient) conditions:
 - *Circular wait*: There is at least one (or more) cycles in the resource allocation graph
 - two or more processes are waiting for a resource held by the other

Example Scenario

- (1) Process A requests and is granted exclusive access to CD-ROM (Resource 1)
- (2) Process B requests and is granted exclusive access to printer (Resource 2)
- (3) Process A requests access to printer (Resource 2), but is blocked
- (4) Process B requests access to CD-ROM (Resource 1), but is blocked



Deadlock Prevention

- Prevent deadlock by preventing any one of the conditions
 - Prevent mutual exclusion
 - Basic property of resources which can only effect minor usage modifications
 - Minimise the extent of mutual exclusive access for a resource
 - e.g. only lock out parts of database that need to be modified (records)
 - Prevent hold-and-wait
 - Force processes to request all required resources initially, otherwise block
 - *Problem 1*: Process may block a long time waiting for all resources to be available
 - *Problem 2*: Resources allocated to a process may remain unused for a long time
 - *Problem 3*: Process may not know how many resources it requires before they are needed
 - Prevent no preemption
 - Basic property of resource which cannot be changed other than requiring process to **checkpoint, rollback** and **recover** when such a resource is forcibly removed.
 - Prevent circular wait
 - Define linear ordering on resources and allow a process to acquire only resources which are subsequent to resources already acquired:
 - Example Scenario, step(4): Process B holds Resource 2 and is thereby prevented from attempting to acquire a lower order resource (Resource 1), it must continue in a similar fashion (requesting other resources of higher order) or terminate (and release Resource 2 for process A), it cannot otherwise block.
 - Unnecessarily restrictive and inefficient

Deadlock Detection and Recovery

- Detection of deadlocks
 - Resources are allowed to be acquired without restriction
 - OS periodically checks for the occurrence of deadlocks
 - detecting deadlocks is not easy and time consuming
 - the act of detecting deadlock may create or add to a deadlock condition
 - e.g. if CPU or memory are part of deadlock, attempting to use them in order to run the detection algorithm will simply add to the deadlock (and perhaps lock up the system!)
 - If deadlock is detected, OS must then choose the least expensive way to recover
- Recovery from deadlocks
 - Various options to recover, all of them fatal to one or more processes
 - abort all deadlocked processes → easy but extreme!
 - backup each deadlocked process and re-run hoping deadlock doesn't re-occur → requires checkpoint, rollback and recovery mechanisms: safe, but not efficient
 - successively abort each deadlocked process until deadlock no longer exists → select processes which have run the least time or produced the least output
 - successively preempt resources until deadlock no longer exists → select resources which are held by the least number of processes

Deadlock Avoidance (*)

- Banker's Algorithm (Dijkstra in 1965)
 - Define the state of a system
 - List of processes
 - number and type of resources used
 - maximum number of resources of each type that can be held
 - Number of resources of each type that are still available
 - Safe, Unsafe and Deadlock states
 - *Safe*: OS can allocate remaining available resources in such a way that all possible requests will be satisfied and no process will block
 - *Unsafe*: A particular sequence of process requests exist which can lead to a potential deadlock no matter what the OS does
 - *Deadlock*: No more resources are available and all processes have been blocked
 - Allocate resources so that the state of the system is never unsafe
 - Algorithm has to examine all subsequent requests in order to see if the system can ever get into a deadlock state → if so request will make system unsafe
 - If process attempts to make an unsafe request it is blocked (even though the resource it wants is free)

Deadlock Avoidance

- Single resource class case

Name	Used	Maximum
Andy	0	6
Barbara	0	5
Marvin	0	4
Suzanne	0	7

Available: 10

(a)

Name	Used	Maximum
Andy	1	6
Barbara	1	5
Marvin	2	4
Suzanne	4	7

Available: 2

(b)

Name	Used	Maximum
Andy	1	6
Barbara	2	5
Marvin	2	4
Suzanne	4	7

Available: 1

(c)

• Figure 3.11, osdi2: (a) initial state (b) safe state (c) unsafe state

- safe state (b)
 - OS can allow Marvin to complete (since Marvin can request the maximum number of resources), Marvin then releases all 4 of his resources when finished, then Suzanne or Barbara can be allowed to complete, and so on.
 - If OS allows Barbara a request → unsafe state (c)!
- unsafe state (c)
 - If any one customer (process) requests the maximum number of resources there will not be enough and deadlock will arise
- Multiple resource class case is more complicated to analyse

Ignoring Deadlocks

- Computer Scientist at University (also COS214 lecture!)
 - Will always attempt to address the possibility of system deadlocks
 - Deadlock prevention strategies
 - Deadlock detection and recovery mechanisms
 - Deadlock avoidance using the Banker's algorithm
- IT engineer in the Real World (post-final year COS214 students)
 - What is the cost of implementing a deadlock strategy?
 - Development costs (time and labour costs in developing a strategy)
 - System costs (increased system costs in implementing strategy)
 - Maintenance costs (debugging, enhancing and updating deadlock strategy)
 - What is the cost of not bothering to implement a deadlock strategy?
 - How many times will a deadlock actually occur in the lifetime of the system?
 - How fatal is the deadlock?
 - *Example:* Operating System resource limits → Process Table
 - If process table runs out of space then system may deadlock if processes can only be terminated by needing to spawn child processes (e.g. the “kill” process)
 - Solution 1? If OS can not support more than 50 processes and no more than 25 users at any one time, then we limit each user to no more than 2 processes!!
 - Solution 2? Ignore problem, allow users to create as many processes as they like and the few times the system is that busy → reboot or buy more memory!